Decision trees, Random Forests, Bagging and Boosting

Morten Hjorth-Jensen[1,2]

[1]Department of Physics, University of Oslo
[2]Department of Physics and Astronomy and National Superconducting Cyclotron Laboratory, Michigan State University

Feb 14, 2021

Decision trees, overarching aims

We start here with the most basic algorithm, the so-called decision tree. With this basic algorithm we can in turn build more complex networks, spanning from homogeneous and heterogenous forests (bagging, random forests and more) to one of the most popular supervised algorithms nowadays, the extreme gradient boosting, or just XGBoost. But let us start with the simplest possible ingredient.

Decision trees are supervised learning algorithms used for both, classification and regression tasks.

The main idea of decision trees is to find those descriptive features which contain the most **information** regarding the target feature and then split the dataset along the values of these features such that the target feature values for the resulting underlying datasets are as pure as possible.

The descriptive features which reproduce best the target/output features are normally said to be the most informative ones. The process of finding the **most informative** feature is done until we accomplish a stopping criteria where we then finally end up in so called **leaf nodes**.

Basics of a tree

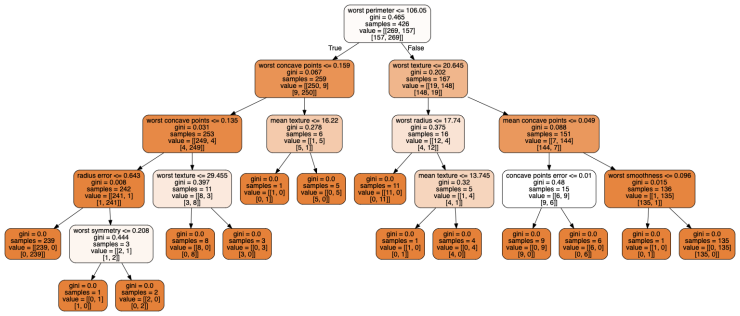
A decision tree is typically divided into a **root node**, the **interior nodes**, and the final **leaf nodes** or just **leaves**. These entities are then connected by so-called **branches**.

The leaf nodes contain the predictions we will make for new query instances presented to our trained model. This is possible since the model has learned the underlying structure of the training data and hence can, given some assumptions, make predictions about the target feature value (class) of unseen query instances.

A Sketch of a Tree, Regression problem

A Sketch of a Tree, Classification problem

A typical Decision Tree with its pertinent Jargon, Classification Problem



This tree was produced using the Wisconsin cancer data (discussed here as well, see code examples below) using **Scikit-Learn**'s decision tree classifier. Here we have used the so-called **gini** index (see below) to split the various branches.

General Features

The overarching approach to decision trees is a top-down approach.

- A leaf provides the classification of a given instance.
- A node specifies a test of some attribute of the instance.
- A branch corresponds to a possible values of an attribute.
- An instance is classified by starting at the root node of the tree, testing the attribute specified by this node, then moving down the tree branch corresponding to the value of the attribute in the given example.

This process is then repeated for the subtree rooted at the new node.

How do we set it up?

In simplified terms, the process of training a decision tree and predicting the target features of query instances is as follows:

1. Present a dataset containing of a number of training instances characterized by a number of descriptive features and a target feature

2. Train the decision tree model by continuously splitting the target feature along the values of the descriptive features using a measure of information gain during the training process

3. Grow the tree until we accomplish a stopping criteria create leaf nodes which represent the *predictions* we want to make for new query instances

4. Show query instances to the tree and run down the tree until we arrive at leaf nodes

Then we are essentially done!

Decision trees and Regression

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression

steps=250

distance=0
x=0
distance_list=[]
steps_list=[]
while x<steps:
    distance+=np.random.randint(-1,2)
    distance_list.append(distance)
    x+=1
    steps_list.append(x)
plt.plot(steps_list,distance_list, color='green', label="Random Walk Data")

steps_list=np.asarray(steps_list)
distance_list=np.asarray(distance_list)

X=steps_list[:,np.newaxis]

#Polynomial fits

#Degree 2
poly_features=PolynomialFeatures(degree=2, include_bias=False)
X_poly=poly_features.fit_transform(X)

lin_reg=LinearRegression()
poly_fit=lin_reg.fit(X_poly,distance_list)
b=lin_reg.coef_
c=lin_reg.intercept_
print ("2nd degree coefficients:")
print ("zero power: ",c)
print ("first power: ", b[0])
print ("second power: ",b[1])

z = np.arange(0, steps, .01)
z_mod=b[1]*z**2+b[0]*z+c

fit_mod=b[1]*X**2+b[0]*X+c
plt.plot(z, z_mod, color='r', label="2nd Degree Fit")
plt.title("Polynomial Regression")
```

```
plt.xlabel("Steps")
plt.ylabel("Distance")

#Degree 10
poly_features10=PolynomialFeatures(degree=10, include_bias=False)
X_poly10=poly_features10.fit_transform(X)

poly_fit10=lin_reg.fit(X_poly10,distance_list)

y_plot=poly_fit10.predict(X_poly10)
plt.plot(X, y_plot, color='black', label="10th Degree Fit")

plt.legend()
plt.show()


#Decision Tree Regression
from sklearn.tree import DecisionTreeRegressor
regr_1=DecisionTreeRegressor(max_depth=2)
regr_2=DecisionTreeRegressor(max_depth=5)
regr_3=DecisionTreeRegressor(max_depth=7)
regr_1.fit(X, distance_list)
regr_2.fit(X, distance_list)
regr_3.fit(X, distance_list)

X_test = np.arange(0.0, steps, 0.01)[:, np.newaxis]
y_1 = regr_1.predict(X_test)
y_2 = regr_2.predict(X_test)
y_3=regr_3.predict(X_test)

# Plot the results
plt.figure()
plt.scatter(X, distance_list, s=2.5, c="black", label="data")
plt.plot(X_test, y_1, color="red",
         label="max_depth=2", linewidth=2)
plt.plot(X_test, y_2, color="green", label="max_depth=5", linewidth=2)
plt.plot(X_test, y_3, color="m", label="max_depth=7", linewidth=2)

plt.xlabel("Data")
plt.ylabel("Darget")
plt.title("Decision Tree Regression")
plt.legend()
plt.show()
```

Building a tree, regression

There are mainly two steps

1. We split the predictor space (the set of possible values $x_1, x_2, \ldots, x_p$) into J distinct and non-non-overlapping regions, $R_1, R_2, \ldots, R_J$.

2. For every observation that falls into the region R_j , we make the same prediction, which is simply the mean of the response values for the training observations in R_j.

How do we construct the regions $R_1, \ldots, R_J$? In theory, the regions could have any shape. However, we choose to divide the predictor space into high-

dimensional rectangles, or boxes, for simplicity and for ease of interpretation of the resulting predictive model. The goal is to find boxes $R_1, \ldots, R_J$ that minimize the MSE, given by

$$\sum_{j=1}^{J} \sum_{i \in R_j} (y_i - \overline{y}_{R_j})^2,$$

where $\overline{y}_{R_j}$ is the mean response for the training observations within box j.

A top-down approach, recursive binary splitting

Unfortunately, it is computationally infeasible to consider every possible partition of the feature space into J boxes. The common strategy is to take a top-down approach

The approach is top-down because it begins at the top of the tree (all observations belong to a single region) and then successively splits the predictor space; each split is indicated via two new branches further down on the tree. It is greedy because at each step of the tree-building process, the best split is made at that particular step, rather than looking ahead and picking a split that will lead to a better tree in some future step.

Making a tree

In order to implement the recursive binary splitting we start by selecting the predictor x_j and a cutpoint s that splits the predictor space into two regions R_1 and R_2

$$\{X | x_j < s\},$$

and

$$\{X | x_j \geq s\},$$

so that we obtain the lowest MSE, that is

$$\sum_{i:x_i \in R_j} (y_i - \overline{y}_{R_1})^2 + \sum_{i:x_i \in R_2} (y_i - \overline{y}_{R_2})^2,$$

which we want to minimize by considering all predictors $x_1, x_2, \ldots, x_p$. We consider also all possible values of s for each predictor. These values could be determined by randomly assigned numbers or by starting at the midpoint and then proceed till we find an optimal value.

For any j and s, we define the pair of half-planes where $\overline{y}_{R_1}$ is the mean response for the training observations in $R_1(j, s)$, and $\overline{y}_{R_2}$ is the mean response for the training observations in $R_2(j, s)$.

Finding the values of j and s that minimize the above equation can be done quite quickly, especially when the number of features p is not too large.

Next, we repeat the process, looking for the best predictor and best cutpoint in order to split the data further so as to minimize the MSE within each of the

resulting regions. However, this time, instead of splitting the entire predictor space, we split one of the two previously identified regions. We now have three regions. Again, we look to split one of these three regions further, so as to minimize the MSE. The process continues until a stopping criterion is reached; for instance, we may continue until no region contains more than five observations.

Pruning the tree

The above procedure is rather straightforward, but leads often to overfitting and unnecessarily large and complicated trees. The basic idea is to grow a large tree T_0 and then prune it back in order to obtain a subtree. A smaller tree with fewer splits (fewer regions) can lead to smaller variance and better interpretation at the cost of a little more bias.

The so-called Cost complexity pruning algorithm gives us a way to do just this. Rather than considering every possible subtree, we consider a sequence of trees indexed by a nonnegative tuning parameter α.

Read more at the following Scikit-Learn link on pruning.

Cost complexity pruning

For each value of α there corresponds a subtree $T \in T_0$ such that

$$\sum_{m=1}^{\overline{T}} \sum_{i:x_i \in R_m} (y_i - \overline{y}_{R_m})^2 + \alpha \overline{T},$$

is as small as possible. Here $\overline{T}$ is the number of terminal nodes of the tree T , R_m is the rectangle (i.e. the subset of predictor space) corresponding to the m-th terminal node.

The tuning parameter α controls a trade-off between the subtree's complexity and its fit to the training data. When $\alpha = 0$, then the subtree T will simply equal T_0, because then the above equation just measures the training error. However, as α increases, there is a price to pay for having a tree with many terminal nodes. The above equation will tend to be minimized for a smaller subtree.

It turns out that as we increase α from zero branches get pruned from the tree in a nested and predictable fashion, so obtaining the whole sequence of subtrees as a function of α is easy. We can select a value of α using a validation set or using cross-validation. We then return to the full data set and obtain the subtree corresponding to α.

Schematic Regression Procedure

Building a Regression Tree.

1. Use recursive binary splitting to grow a large tree on the training data, stopping only when each terminal node has fewer than some minimum number of observations.

2. Apply cost complexity pruning to the large tree in order to obtain a sequence of best subtrees, as a function of α.

3. Use for example K-fold cross-validation to choose α. Divide the training observations into K folds. For each $k = 1, 2, \ldots, K$ we:

   - repeat steps 1 and 2 on all but the k-th fold of the training data.
   - Then we valuate the mean squared prediction error on the data in the left-out k-th fold, as a function of α.
   - Finally we average the results for each value of α, and pick α to minimize the average error.

4. Return the subtree from Step 2 that corresponds to the chosen value of α.

A Classification Tree

A classification tree is very similar to a regression tree, except that it is used to predict a qualitative response rather than a quantitative one. Recall that for a regression tree, the predicted response for an observation is given by the mean response of the training observations that belong to the same terminal node. In contrast, for a classification tree, we predict that each observation belongs to the most commonly occurring class of training observations in the region to which it belongs. In interpreting the results of a classification tree, we are often interested not only in the class prediction corresponding to a particular terminal node region, but also in the class proportions among the training observations that fall into that region.

Growing a classification tree

The task of growing a classification tree is quite similar to the task of growing a regression tree. Just as in the regression setting, we use recursive binary splitting to grow a classification tree. However, in the classification setting, the MSE cannot be used as a criterion for making the binary splits. A natural alternative to MSE is the **classification error rate**. Since we plan to assign an observation in a given region to the most commonly occurring error rate class of training observations in that region, the classification error rate is simply the fraction of the training observations in that region that do not belong to the most common class.

When building a classification tree, either the Gini index or the entropy are typically used to evaluate the quality of a particular split, since these two approaches are more sensitive to node purity than is the classification error rate.

Classification tree, how to split nodes

If our targets are the outcome of a classification process that takes for example $k = 1, 2, \dots, K$ values, the only thing we need to think of is to set up the splitting criteria for each node.

We define a PDF p_{mk} that represents the number of observations of a class k in a region R_m with N_m observations. We represent this likelihood function in terms of the proportion $I(y_i = k)$ of observations of this class in the region R_m as

$$p_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i = k).$$

We let p_{mk} represent the majority class of observations in region m. The three most common ways of splitting a node are given by

- Misclassification error

$$p_{mk} = \frac{1}{N_m} \sum_{x_i \in R_m} I(y_i \neq k) = 1 - p_{mk}.$$

- Gini index g

$$g = \sum_{k=1}^{K} p_{mk}(1 - p_{mk}).$$

- Information entropy or just entropy s

$$s = -\sum_{k=1}^{K} p_{mk} \log p_{mk}.$$

Visualizing the Tree, Classification

```
import os
from sklearn.datasets import load_breast_cancer
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import confusion_matrix
from sklearn.tree import export_graphviz

from IPython.display import Image
from pydot import graph_from_dot_data
import pandas as pd
import numpy as np


cancer = load_breast_cancer()
X = pd.DataFrame(cancer.data, columns=cancer.feature_names)
print(X)
y = pd.Categorical.from_codes(cancer.target, cancer.target_names)
y = pd.get_dummies(y)
print(y)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=1)
```

```
tree_clf = DecisionTreeClassifier(max_depth=5)
tree_clf.fit(X_train, y_train)

export_graphviz(
    tree_clf,
    out_file="DataFiles/cancer.dot",
    feature_names=cancer.feature_names,
    class_names=cancer.target_names,
    rounded=True,
    filled=True
)
cmd = 'dot -Tpng DataFiles/cancer.dot -o DataFiles/cancer.png'
os.system(cmd)
```

Visualizing the Tree, The Moons

```
# Common imports
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_moons
from sklearn.tree import export_graphviz
from pydot import graph_from_dot_data
import pandas as pd
import os

np.random.seed(42)
X, y = make_moons(n_samples=100, noise=0.25, random_state=53)
X_train, X_test, y_train, y_test = train_test_split(X,y,random_state=0)
tree_clf = DecisionTreeClassifier(max_depth=5)
tree_clf.fit(X_train, y_train)

export_graphviz(
    tree_clf,
    out_file="DataFiles/moons.dot",
    rounded=True,
    filled=True
)
cmd = 'dot -Tpng DataFiles/moons.dot -o DataFiles/moons.png'
os.system(cmd)
```

Other ways of visualizing the trees

Scikit-Learn has also another way to visualize the trees which is very useful, here with the Iris data.

```
from sklearn.datasets import load_iris
from sklearn import tree
X, y = load_iris(return_X_y=True)
tree_clf = tree.DecisionTreeClassifier()
tree_clf = tree_clf.fit(X, y)
# and then plot the tree
tree.plot_tree(tree_clf)
```

Printing out as text

Alternatively, the tree can also be exported in textual format with the function exporttext. This method doesn't require the installation of external libraries and is more compact:

```
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.tree import export_text
iris = load_iris()
decision_tree = DecisionTreeClassifier(random_state=0, max_depth=2)
decision_tree = decision_tree.fit(iris.data, iris.target)
r = export_text(decision_tree, feature_names=iris['feature_names'])
print(r)
```

Algorithms for Setting up Decision Trees

Two algorithms stand out in the set up of decision trees:

1. The CART (Classification And Regression Tree) algorithm for both classification and regression

2. The ID3 algorithm based on the computation of the information gain for classification

We discuss both algorithms with applications here. The popular library **Scikit-Learn** uses the CART algorithm. For classification problems you can use either the **gini** index or the **entropy** to split a tree in two branches.

The CART algorithm for Classification

For classification, the CART algorithm splits the data set in two subsets using a single feature k and a threshold t_k. This could be for example a threshold set by a number below a certain circumference of a malign tumor.

How do we find these two quantities? We search for the pair (k, t_k) that produces the purest subset using for example the **gini** factor G. The cost function it tries to minimize is then

$$C(k, t_k) = \frac{m_{\text{left}}}{m} G_{\text{left}} + \frac{m_{\text{right}}}{m} G_{\text{right}},$$

where $G_{\text{left/right}}$ measures the impurity of the left/right subset and $m_{\text{left/right}}$ is the number of instances in the left/right subset

Once it has successfully split the training set in two, it splits the subsets using the same logic, then the subsubsets and so on, recursively. It stops recursing once it reaches the maximum depth (defined by the max_depth hyperparameter), or if it cannot find a split that will reduce impurity. A few other hyperparameters control additional stopping conditions such as the $min\_samples\_split$, $min\_samples\_leaf$, $min\_weight\_fraction\_leaf$, and max_leaf_nodes.

The CART algorithm for Regression

The CART algorithm for regression works is similar to the one for classification except that instead of trying to split the training set in a way that minimizes say the **gini** or **entropy** impurity, it now tries to split the training set in a way that minimizes our well-known mean-squared error (MSE). The cost function is now

$$C(k, t_k) = \frac{m_{\text{left}}}{m}\text{MSE}_{\text{left}} + \frac{m_{\text{right}}}{m}\text{MSE}_{\text{right}}.$$

Here the MSE for a specific node is defined as

$$\text{MSE}_{\text{node}} = \frac{1}{m_{\text{node}}}\sum_{i\in\text{node}}(\overline{y}_{\text{node}} - y_i)^2,$$

with

$$\overline{y}_{\text{node}} = \frac{1}{m_{\text{node}}}\sum_{i\in\text{node}} y_i,$$

the mean value of all observations in a specific node.

Without any regularization, the regression task for decision trees, just like for classification tasks, is prone to overfitting.

Cancer Data again now with Decision Trees and other Methods

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier

# Load the data
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_state=0)
print(X_train.shape)
print(X_test.shape)
# Logistic Regression
logreg = LogisticRegression(solver='lbfgs')
logreg.fit(X_train, y_train)
print("Test set accuracy with Logistic Regression: {:.2f}".format(logreg.score(X_test,y_test)))
# Support vector machine
svm = SVC(gamma='auto', C=100)
svm.fit(X_train, y_train)
print("Test set accuracy with SVM: {:.2f}".format(svm.score(X_test,y_test)))
# Decision Trees
deep_tree_clf = DecisionTreeClassifier(max_depth=None)
deep_tree_clf.fit(X_train, y_train)
print("Test set accuracy with Decision Trees: {:.2f}".format(deep_tree_clf.score(X_test,y_test)))
#now scale the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
```

```
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Logistic Regression
logreg.fit(X_train_scaled, y_train)
print("Test set accuracy Logistic Regression with scaled data: {:.2f}".format(logreg.score(X_test_
# Support Vector Machine
svm.fit(X_train_scaled, y_train)
print("Test set accuracy SVM with scaled data: {:.2f}".format(logreg.score(X_test_scaled,y_test)))
# Decision Trees
deep_tree_clf.fit(X_train_scaled, y_train)
print("Test set accuracy with Decision Trees and scaled data: {:.2f}".format(deep_tree_clf.score(X
```

Another example, the moons again

```
from __future__ import division, print_function, unicode_literals

# Common imports
import numpy as np
import os

# to make this notebook's output stable across runs
np.random.seed(42)

# To plot pretty figures
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12


from sklearn.svm import SVC
from sklearn import datasets
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import make_moons
from sklearn.tree import export_graphviz

Xm, ym = make_moons(n_samples=100, noise=0.25, random_state=53)

deep_tree_clf1 = DecisionTreeClassifier(random_state=42)
deep_tree_clf2 = DecisionTreeClassifier(min_samples_leaf=4, random_state=42)
deep_tree_clf1.fit(Xm, ym)
deep_tree_clf2.fit(Xm, ym)


def plot_decision_boundary(clf, X, y, axes=[0, 7.5, 0, 3], iris=True, legend=False, plot_training=
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)
    custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])
    plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=custom_cmap)
    if not iris:
        custom_cmap2 = ListedColormap(['#7d7d58','#4c4c7f','#507d50'])
        plt.contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8)
    if plot_training:
        plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", label="Iris-Setosa")
```

```
            plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", label="Iris-Versicolor")
            plt.plot(X[:, 0][y==2], X[:, 1][y==2], "g^", label="Iris-Virginica")
            plt.axis(axes)
        if iris:
            plt.xlabel("Petal length", fontsize=14)
            plt.ylabel("Petal width", fontsize=14)
        else:
            plt.xlabel(r"$x_1$", fontsize=18)
            plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
        if legend:
            plt.legend(loc="lower right", fontsize=14)
    plt.figure(figsize=(11, 4))
    plt.subplot(121)
    plot_decision_boundary(deep_tree_clf1, Xm, ym, axes=[-1.5, 2.5, -1, 1.5], iris=False)
    plt.title("No restrictions", fontsize=16)
    plt.subplot(122)
    plot_decision_boundary(deep_tree_clf2, Xm, ym, axes=[-1.5, 2.5, -1, 1.5], iris=False)
    plt.title("min_samples_leaf = {}".format(deep_tree_clf2.min_samples_leaf), fontsize=14)
    plt.show()
```

Playing around with regions

```
np.random.seed(6)
Xs = np.random.rand(100, 2) - 0.5
ys = (Xs[:, 0] > 0).astype(np.float32) * 2

angle = np.pi/4
rotation_matrix = np.array([[np.cos(angle), -np.sin(angle)], [np.sin(angle), np.cos(angle)]])
Xsr = Xs.dot(rotation_matrix)

tree_clf_s = DecisionTreeClassifier(random_state=42)
tree_clf_s.fit(Xs, ys)
tree_clf_sr = DecisionTreeClassifier(random_state=42)
tree_clf_sr.fit(Xsr, ys)

plt.figure(figsize=(11, 4))
plt.subplot(121)
plot_decision_boundary(tree_clf_s, Xs, ys, axes=[-0.7, 0.7, -0.7, 0.7], iris=False)
plt.subplot(122)
plot_decision_boundary(tree_clf_sr, Xsr, ys, axes=[-0.7, 0.7, -0.7, 0.7], iris=False)

plt.show()
```

Regression trees

```
# Quadratic training set + noise
np.random.seed(42)
m = 200
X = np.random.rand(m, 1)
y = 4 * (X - 0.5) ** 2
y = y + np.random.randn(m, 1) / 10

from sklearn.tree import DecisionTreeRegressor

tree_reg = DecisionTreeRegressor(max_depth=2, random_state=42)
tree_reg.fit(X, y)
```

Final regressor code

```
from sklearn.tree import DecisionTreeRegressor

tree_reg1 = DecisionTreeRegressor(random_state=42, max_depth=2)
tree_reg2 = DecisionTreeRegressor(random_state=42, max_depth=3)
tree_reg1.fit(X, y)
tree_reg2.fit(X, y)

def plot_regression_predictions(tree_reg, X, y, axes=[0, 1, -0.2, 1], ylabel="$y$"):
    x1 = np.linspace(axes[0], axes[1], 500).reshape(-1, 1)
    y_pred = tree_reg.predict(x1)
    plt.axis(axes)
    plt.xlabel("$x_1$", fontsize=18)
    if ylabel:
        plt.ylabel(ylabel, fontsize=18, rotation=0)
    plt.plot(X, y, "b.")
    plt.plot(x1, y_pred, "r.-", linewidth=2, label=r"$\hat{y}$")

plt.figure(figsize=(11, 4))
plt.subplot(121)
plot_regression_predictions(tree_reg1, X, y)
for split, style in ((0.1973, "k-"), (0.0917, "k--"), (0.7718, "k--")):
    plt.plot([split, split], [-0.2, 1], style, linewidth=2)
plt.text(0.21, 0.65, "Depth=0", fontsize=15)
plt.text(0.01, 0.2, "Depth=1", fontsize=13)
plt.text(0.65, 0.8, "Depth=1", fontsize=13)
plt.legend(loc="upper center", fontsize=18)
plt.title("max_depth=2", fontsize=14)

plt.subplot(122)
plot_regression_predictions(tree_reg2, X, y, ylabel=None)
for split, style in ((0.1973, "k-"), (0.0917, "k--"), (0.7718, "k--")):
    plt.plot([split, split], [-0.2, 1], style, linewidth=2)
for split in (0.0458, 0.1298, 0.2873, 0.9040):
    plt.plot([split, split], [-0.2, 1], "k:", linewidth=1)
plt.text(0.3, 0.5, "Depth=2", fontsize=13)
plt.title("max_depth=3", fontsize=14)

plt.show()

tree_reg1 = DecisionTreeRegressor(random_state=42)
tree_reg2 = DecisionTreeRegressor(random_state=42, min_samples_leaf=10)
tree_reg1.fit(X, y)
tree_reg2.fit(X, y)

x1 = np.linspace(0, 1, 500).reshape(-1, 1)
y_pred1 = tree_reg1.predict(x1)
y_pred2 = tree_reg2.predict(x1)

plt.figure(figsize=(11, 4))

plt.subplot(121)
plt.plot(X, y, "b.")
plt.plot(x1, y_pred1, "r.-", linewidth=2, label=r"$\hat{y}$")
plt.axis([0, 1, -0.2, 1.1])
plt.xlabel("$x_1$", fontsize=18)
plt.ylabel("$y$", fontsize=18, rotation=0)
plt.legend(loc="upper center", fontsize=18)
plt.title("No restrictions", fontsize=14)
```

```
plt.subplot(122)
plt.plot(X, y, "b.")
plt.plot(x1, y_pred2, "r.-", linewidth=2, label=r"$\hat{y}$")
plt.axis([0, 1, -0.2, 1.1])
plt.xlabel("$x_1$", fontsize=18)
plt.title("min_samples_leaf={}".format(tree_reg2.min_samples_leaf), fontsize=14)

plt.show()
```

Pros and cons of trees, pros

- White box, easy to interpret model. Some people believe that decision trees more closely mirror human decision-making than do the regression and classification approaches discussed earlier (think of support vector machines)
- Trees are very easy to explain to people. In fact, they are even easier to explain than linear regression!
- No feature normalization needed
- Tree models can handle both continuous and categorical data (Classification and Regression Trees)
- Can model nonlinear relationships
- Can model interactions between the different descriptive features
- Trees can be displayed graphically, and are easily interpreted even by a non-expert (especially if they are small)

Disadvantages

- Unfortunately, trees generally do not have the same level of predictive accuracy as some of the other regression and classification approaches
- If continuous features are used the tree may become quite large and hence less interpretable
- Decision trees are prone to overfit the training data and hence do not well generalize the data if no stopping criteria or improvements like pruning, boosting or bagging are implemented
- Small changes in the data may lead to a completely different tree. This issue can be addressed by using ensemble methods like bagging, boosting or random forests
- Unbalanced datasets where some target feature values occur much more frequently than others may lead to biased trees since the frequently occurring feature values are preferred over the less frequently occurring ones.

- If the number of features is relatively large (high dimensional) and the number of instances is relatively low, the tree might overfit the data
- Features with many levels may be preferred over features with less levels since for them it is *more easy* to split the dataset such that the sub datasets only contain pure target feature values. This issue can be addressed by preferring for instance the information gain ratio as splitting criteria over information gain

However, by aggregating many decision trees, using methods like bagging, random forests, and boosting, the predictive performance of trees can be substantially improved.

Ensemble Methods: From a Single Tree to Many Trees and Extreme Boosting, Meet the Jungle of Methods

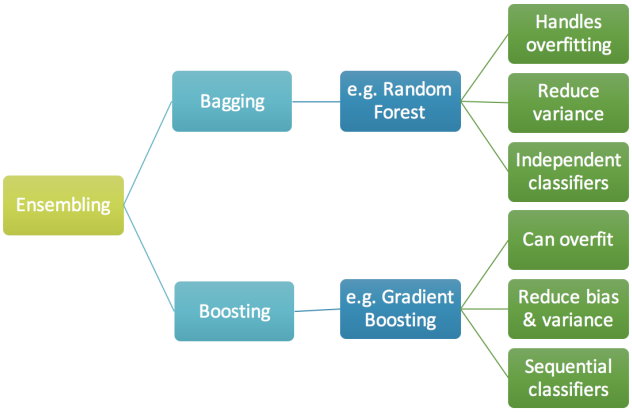
As stated above and seen in many of the examples discussed here about a single decision tree, we often end up overfitting our training data. This normally means that we have a high variance. Can we reduce the variance of a statistical learning method?

This leads us to a set of different methods that can combine different machine learning algorithms or just use one of them to construct forests and jungles of trees, homogeneous ones or heterogenous ones. These methods are recognized by different names which we will try to explain here. These are

1. Voting classifiers
2. Bagging and Pasting
3. Random forests
4. Boosting methods, from adaptive to Extreme Gradient Boosting (XGBoost)

We discuss these methods here.

An Overview of Ensemble Methods



Bagging

The **plain** decision trees suffer from high variance. This means that if we split the training data into two parts at random, and fit a decision tree to both halves, the results that we get could be quite different. In contrast, a procedure with low variance will yield similar results if applied repeatedly to distinct data sets; linear regression tends to have low variance, if the ratio of n to p is moderately large.

Bootstrap aggregation, or just **bagging**, is a general-purpose procedure for reducing the variance of a statistical learning method.

More bagging

Bagging typically results in improved accuracy over prediction using a single tree. Unfortunately, however, it can be difficult to interpret the resulting model. Recall that one of the advantages of decision trees is the attractive and easily interpreted diagram that results.

However, when we bag a large number of trees, it is no longer possible to represent the resulting statistical learning procedure using a single tree, and it is no longer clear which variables are most important to the procedure. Thus, bagging improves prediction accuracy at the expense of interpretability. Although the collection of bagged trees is much more difficult to interpret than a single tree, one can obtain an overall summary of the importance of each predictor using the MSE (for bagging regression trees) or the Gini index (for bagging classification trees). In the case of bagging regression trees, we can record the total amount that the MSE is decreased due to splits over a given predictor, averaged over all

B possible trees. A large value indicates an important predictor. Similarly, in the context of bagging classification trees, we can add up the total amount that the Gini index is decreased by splits over a given predictor, averaged over all B trees.

Simple Voting Example, head or tail

```
heads_proba = 0.51
coin_tosses = (np.random.rand(10000, 10) < heads_proba).astype(np.int32)
cumulative_heads_ratio = np.cumsum(coin_tosses, axis=0) / np.arange(1, 10001).reshape(-1, 1)
plt.figure(figsize=(8,3.5))
plt.plot(cumulative_heads_ratio)
plt.plot([0, 10000], [0.51, 0.51], "k--", linewidth=2, label="51%")
plt.plot([0, 10000], [0.5, 0.5], "k-", label="50%")
plt.xlabel("Number of coin tosses")
plt.ylabel("Heads ratio")
plt.legend(loc="lower right")
plt.axis([0, 10000, 0.42, 0.58])
save_fig("votingsimple")
plt.show()
```

Using the Voting Classifier

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(solver="liblinear", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
svm_clf = SVC(gamma="auto", random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')

voting_clf.fit(X_train, y_train)

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))

log_clf = LogisticRegression(solver="liblinear", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
svm_clf = SVC(gamma="auto", probability=True, random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='soft')
```

```
voting_clf.fit(X_train, y_train)

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

Please, not the moons again! Voting and Bagging

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(random_state=42)
rnd_clf = RandomForestClassifier(random_state=42)
svm_clf = SVC(random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))

log_clf = LogisticRegression(random_state=42)
rnd_clf = RandomForestClassifier(random_state=42)
svm_clf = SVC(probability=True, random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='soft')
voting_clf.fit(X_train, y_train)

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

Bagging Examples

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

bag_clf = BaggingClassifier(
```

```
    DecisionTreeClassifier(random_state=42), n_estimators=500,
    max_samples=100, bootstrap=True, n_jobs=-1, random_state=42)
bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)

from sklearn.metrics import accuracy_score
print(accuracy_score(y_test, y_pred))

tree_clf = DecisionTreeClassifier(random_state=42)
tree_clf.fit(X_train, y_train)
y_pred_tree = tree_clf.predict(X_test)
print(accuracy_score(y_test, y_pred_tree))

from matplotlib.colors import ListedColormap

def plot_decision_boundary(clf, X, y, axes=[-1.5, 2.5, -1, 1.5], alpha=0.5, contour=True):
    x1s = np.linspace(axes[0], axes[1], 100)
    x2s = np.linspace(axes[2], axes[3], 100)
    x1, x2 = np.meshgrid(x1s, x2s)
    X_new = np.c_[x1.ravel(), x2.ravel()]
    y_pred = clf.predict(X_new).reshape(x1.shape)
    custom_cmap = ListedColormap(['#fafab0','#9898ff','#a0faa0'])
    plt.contourf(x1, x2, y_pred, alpha=0.3, cmap=custom_cmap)
    if contour:
        custom_cmap2 = ListedColormap(['#7d7d58','#4c4c7f','#507d50'])
        plt.contour(x1, x2, y_pred, cmap=custom_cmap2, alpha=0.8)
    plt.plot(X[:, 0][y==0], X[:, 1][y==0], "yo", alpha=alpha)
    plt.plot(X[:, 0][y==1], X[:, 1][y==1], "bs", alpha=alpha)
    plt.axis(axes)
    plt.xlabel(r"$x_1$", fontsize=18)
    plt.ylabel(r"$x_2$", fontsize=18, rotation=0)
plt.figure(figsize=(11,4))
plt.subplot(121)
plot_decision_boundary(tree_clf, X, y)
plt.title("Decision Tree", fontsize=14)
plt.subplot(122)
plot_decision_boundary(bag_clf, X, y)
plt.title("Decision Trees with Bagging", fontsize=14)
save_fig("baggingtree")
plt.show()
```

Making your own Bootstrap: Changing the Level of the Decision Tree

Let us bring up our good old boostrap example from the linear regression lectures. We change the linerar regression algorithm with a decision tree wth different depths and perform a bootstrap aggregate (in this case we perform as many bootstraps as data points n).

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.pipeline import make_pipeline
from sklearn.utils import resample
from sklearn.tree import DecisionTreeRegressor

n = 100
n_boostraps = 100
```

```
maxdepth = 8

# Make data set.
x = np.linspace(-3, 3, n).reshape(-1, 1)
y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2)+ np.random.normal(0, 0.1, x.shape)
error = np.zeros(maxdepth)
bias = np.zeros(maxdepth)
variance = np.zeros(maxdepth)
polydegree = np.zeros(maxdepth)
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2)

from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

# we produce a simple tree first as benchmark
simpletree = DecisionTreeRegressor(max_depth=3)
simpletree.fit(X_train_scaled, y_train)
simpleprediction = simpletree.predict(X_test_scaled)
for degree in range(1,maxdepth):
    model = DecisionTreeRegressor(max_depth=degree)
    y_pred = np.empty((y_test.shape[0], n_boostraps))
    for i in range(n_boostraps):
        x_, y_ = resample(X_train_scaled, y_train)
        model.fit(x_, y_)
        y_pred[:, i] = model.predict(X_test_scaled)#.ravel()

    polydegree[degree] = degree
    error[degree] = np.mean( np.mean((y_test - y_pred)**2, axis=1, keepdims=True) )
    bias[degree] = np.mean( (y_test - np.mean(y_pred, axis=1, keepdims=True))**2 )
    variance[degree] = np.mean( np.var(y_pred, axis=1, keepdims=True) )
    print('Polynomial degree:', degree)
    print('Error:', error[degree])
    print('Bias^2:', bias[degree])
    print('Var:', variance[degree])
    print('{} >= {} + {} = {}'.format(error[degree], bias[degree], variance[degree], bias[degree]+

mse_simpletree= np.mean( np.mean((y_test - simpleprediction)**2)
print(mse_simpletree)
plt.xlim(1,maxdepth)
plt.plot(polydegree, error, label='MSE')
plt.plot(polydegree, bias, label='bias')
plt.plot(polydegree, variance, label='Variance')
plt.legend()
save_fig("baggingboot")
plt.show()
```

Why Voting?

The idea behind boosting, and voting as well can be phrased as follows: **Can a group of people somehow arrive at highly reasoned decisions, despite the weak judgement of the individual members?**

The aim is to create a good classifier by combining several weak classifiers. **A weak classifier is a classifier which is able to produce results that are only slightly better than guessing at random.**

The basic approach is to apply repeatedly (in boosting this is done in an iterative way) a weak classifier to modifications of the data. In voting we simply apply the law of large numbers while in boosting we give more weight to misclassified data in each iteration.

Decision trees play an important role as our weak classifier. They serve as the basic method.

Tossing coins

The simplest case is a so-called voting ensemble. To illustrate this, think of yourself tossing coins with a biased outcome of 51 per cent for heads and 49% for tails. With only few tosses, you may not clearly see this distribution for heads and tails. However, after some thousands of tosses, there will be a clear majority of heads. With 2000 tosses you should see approximately 1020 heads and 980 tails.

We can then state that the outcome is a clear majority of heads. If you do this ten thousand times, it is easy to see that there is a 97% likelihood of a majority of heads.

Another example would be to collect all polls before an election. Different polls may show different likelihoods for a candidate winning with say a majority of the popular vote. The majority vote would then consist in many polls indicating that this candidate will actually win.

The example here shows how we can implement the coin tossing case, clealry demostrating that after some tosses we see the law of large numbers kicking in.

Standard imports first

```
# Common imports
from IPython.display import Image
from pydot import graph_from_dot_data
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn.tree import export_graphviz
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer
from IPython.display import Image
from pydot import graph_from_dot_data
import os

# Where to save the figures and data files
PROJECT_ROOT_DIR = "Results"
FIGURE_ID = "Results/FigureFiles"
DATA_ID = "DataFiles/"

if not os.path.exists(PROJECT_ROOT_DIR):
    os.mkdir(PROJECT_ROOT_DIR)

if not os.path.exists(FIGURE_ID):
    os.makedirs(FIGURE_ID)
```

```
if not os.path.exists(DATA_ID):
    os.makedirs(DATA_ID)

def image_path(fig_id):
    return os.path.join(FIGURE_ID, fig_id)

def data_path(dat_id):
    return os.path.join(DATA_ID, dat_id)

def save_fig(fig_id):
    plt.savefig(image_path(fig_id) + ".png", format='png')
```

Simple Voting Example, head or tail

```
# Common imports
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
plt.rcParams['axes.labelsize'] = 14
plt.rcParams['xtick.labelsize'] = 12
plt.rcParams['ytick.labelsize'] = 12

heads_proba = 0.51
coin_tosses = (np.random.rand(10000, 10) < heads_proba).astype(np.int32)
cumulative_heads_ratio = np.cumsum(coin_tosses, axis=0) / np.arange(1, 10001).reshape(-1, 1)
plt.figure(figsize=(8,3.5))
plt.plot(cumulative_heads_ratio)
plt.plot([0, 10000], [0.51, 0.51], "k--", linewidth=2, label="51%")
plt.plot([0, 10000], [0.5, 0.5], "k-", label="50%")
plt.xlabel("Number of coin tosses")
plt.ylabel("Heads ratio")
plt.legend(loc="lower right")
plt.axis([0, 10000, 0.42, 0.58])
save_fig("votingsimple")
plt.show()
```

Using the Voting Classifier

We can use the voting classifier on other data sets, here the exciting binary case of two distinct objects using the make moons functionality of **Scikit-Learn**.

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)

from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(solver="liblinear", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
svm_clf = SVC(gamma="auto", random_state=42)

voting_clf = VotingClassifier(
```

```
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')

voting_clf.fit(X_train, y_train)

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))

log_clf = LogisticRegression(solver="liblinear", random_state=42)
rnd_clf = RandomForestClassifier(n_estimators=10, random_state=42)
svm_clf = SVC(gamma="auto", probability=True, random_state=42)
voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='soft')
voting_clf.fit(X_train, y_train)

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

Voting and Bagging

```
from sklearn.model_selection import train_test_split
from sklearn.datasets import make_moons

X, y = make_moons(n_samples=500, noise=0.30, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42)
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import VotingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC

log_clf = LogisticRegression(random_state=42)
rnd_clf = RandomForestClassifier(random_state=42)
svm_clf = SVC(random_state=42)

voting_clf = VotingClassifier(
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='hard')
voting_clf.fit(X_train, y_train)

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))

log_clf = LogisticRegression(random_state=42)
rnd_clf = RandomForestClassifier(random_state=42)
svm_clf = SVC(probability=True, random_state=42)

voting_clf = VotingClassifier(
```

```
    estimators=[('lr', log_clf), ('rf', rnd_clf), ('svc', svm_clf)],
    voting='soft')
voting_clf.fit(X_train, y_train)

from sklearn.metrics import accuracy_score

for clf in (log_clf, rnd_clf, svm_clf, voting_clf):
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    print(clf.__class__.__name__, accuracy_score(y_test, y_pred))
```

Random forests

Random forests provide an improvement over bagged trees by way of a small tweak that decorrelates the trees.

As in bagging, we build a number of decision trees on bootstrapped training samples. But when building these decision trees, each time a split in a tree is considered, a random sample of m predictors is chosen as split candidates from the full set of p predictors. The split is allowed to use only one of those m predictors.

A fresh sample of m predictors is taken at each split, and typically we choose

$$m \approx \sqrt{p}.$$

In building a random forest, at each split in the tree, the algorithm is not even allowed to consider a majority of the available predictors.

The reason for this is rather clever. Suppose that there is one very strong predictor in the data set, along with a number of other moderately strong predictors. Then in the collection of bagged variable importance random forest trees, most or all of the trees will use this strong predictor in the top split. Consequently, all of the bagged trees will look quite similar to each other. Hence the predictions from the bagged trees will be highly correlated. Unfortunately, averaging many highly correlated quantities does not lead to as large of a reduction in variance as averaging many uncorrelated quantities. In particular, this means that bagging will not lead to a substantial reduction in variance over a single tree in this setting.

Random Forest Algorithm

The algorithm described here can be applied to both classification and regression problems.

We will grow of forest of say B trees.

1. For $b = 1 : B$
    - Draw a bootstrap sample from the training data organized in our $\boldsymbol{X}$ matrix.
    - We grow then a random forest tree T_b based on the bootstrapped data by repeating the steps outlined till we reach the maximum node size is reached

(a) we select $m \leq p$ variables at random from the p predictors/features

(b) pick the best split point among the m features using for example the CART algorithm and create a new node

(c) split the node into daughter nodes

2. Output then the ensemble of trees $\{T_b\}_1^B$ and make predictions for either a regression type of problem or a classification type of problem.

Random Forests Compared with other Methods on the Cancer Data

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.svm import SVC
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier

# Load the data
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_state=0)
print(X_train.shape)
print(X_test.shape)
# Logistic Regression
logreg = LogisticRegression(solver='lbfgs')
logreg.fit(X_train, y_train)
print("Test set accuracy with Logistic Regression: {:.2f}".format(logreg.score(X_test,y_test)))
# Support vector machine
svm = SVC(gamma='auto', C=100)
svm.fit(X_train, y_train)
print("Test set accuracy with SVM: {:.2f}".format(svm.score(X_test,y_test)))
# Decision Trees
deep_tree_clf = DecisionTreeClassifier(max_depth=None)
deep_tree_clf.fit(X_train, y_train)
print("Test set accuracy with Decision Trees: {:.2f}".format(deep_tree_clf.score(X_test,y_test)))
#now scale the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
# Logistic Regression
logreg.fit(X_train_scaled, y_train)
print("Test set accuracy Logistic Regression with scaled data: {:.2f}".format(logreg.score(X_test_
# Support Vector Machine
svm.fit(X_train_scaled, y_train)
print("Test set accuracy SVM with scaled data: {:.2f}".format(logreg.score(X_test_scaled,y_test)))
# Decision Trees
deep_tree_clf.fit(X_train_scaled, y_train)
print("Test set accuracy with Decision Trees and scaled data: {:.2f}".format(deep_tree_clf.score(X


from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_validate
# Data set not specificied
#Instantiate the model with 500 trees and entropy as splitting criteria
Random_Forest_model = RandomForestClassifier(n_estimators=500,criterion="entropy")
Random_Forest_model.fit(X_train_scaled, y_train)
#Cross validation
accuracy = cross_validate(Random_Forest_model,X_test_scaled,y_test,cv=10)['test_score']
print(accuracy)
print("Test set accuracy with Random Forests and scaled data: {:.2f}".format(Random_Forest_model.s


import scikitplot as skplt
y_pred = Random_Forest_model.predict(X_test_scaled)
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=True)
plt.show()
y_probas = Random_Forest_model.predict_proba(X_test_scaled)
skplt.metrics.plot_roc(y_test, y_probas)
plt.show()
skplt.metrics.plot_cumulative_gain(y_test, y_probas)
plt.show()
```

Recall that the cumulative gains curve shows the percentage of the overall number of cases in a given category *gained* by targeting a percentage of the total number of cases.

Similarly, the receiver operating characteristic curve, or ROC curve, displays the diagnostic ability of a binary classifier system as its discrimination threshold is varied. It plots the true positive rate against the false positive rate.

Compare Bagging on Trees with Random Forests

```
bag_clf = BaggingClassifier(
    DecisionTreeClassifier(splitter="random", max_leaf_nodes=16, random_state=42),
    n_estimators=500, max_samples=1.0, bootstrap=True, n_jobs=-1, random_state=42)

bag_clf.fit(X_train, y_train)
y_pred = bag_clf.predict(X_test)
from sklearn.ensemble import RandomForestClassifier
rnd_clf = RandomForestClassifier(n_estimators=500, max_leaf_nodes=16, n_jobs=-1, random_state=42)
rnd_clf.fit(X_train, y_train)
y_pred_rf = rnd_clf.predict(X_test)
np.sum(y_pred == y_pred_rf) / len(y_pred)
```

Boosting, a Bird's Eye View

The basic idea is to combine weak classifiers in order to create a good classifier. With a weak classifier we often intend a classifier which produces results which are only slightly better than we would get by random guesses.

This is done by applying in an iterative way a weak (or a standard classifier like decision trees) to modify the data. In each iteration we emphasize those observations which are misclassified by weighting them with a factor.

What is boosting? Additive Modelling/Iterative Fitting

Boosting is a way of fitting an additive expansion in a set of elementary basis functions like for example some simple polynomials. Assume for example that we have a function

$$f_M(x) = \sum_{i=1}^{M} \beta_m b(x; \gamma_m),$$

where β_m are the expansion parameters to be determined in a minimization process and $b(x; \gamma_m)$ are some simple functions of the multivariable parameter x which is characterized by the parameters γ_m.

As an example, consider the Sigmoid function we used in logistic regression. In that case, we can translate the function $b(x; \gamma_m)$ into the Sigmoid function

$$\sigma(t) = \frac{1}{1 + \exp(-t)},$$

where $t = \gamma_0 + \gamma_1 x$ and the parameters γ_0 and γ_1 were determined by the Logistic Regression fitting algorithm.

As another example, consider the cost function we defined for linear regression

$$C(\boldsymbol{y}, \boldsymbol{f}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - f(x_i))^2.$$

In this case the function $f(x)$ was replaced by the design matrix $\boldsymbol{X}$ and the unknown linear regression parameters $\boldsymbol{\beta}$, that is $\boldsymbol{f} = \boldsymbol{X}\boldsymbol{\beta}$. In linear regression we can simply invert a matrix and obtain the parameters β by

$$\boldsymbol{\beta} = \left(\boldsymbol{X}^T \boldsymbol{X}\right)^{-1} \boldsymbol{X}^T \boldsymbol{y}.$$

In iterative fitting or additive modeling, we minimize the cost function with respect to the parameters β_m and γ_m.

Iterative Fitting, Regression and Squared-error Cost Function

The way we proceed is as follows (here we specialize to the squared-error cost function)

1. Establish a cost function, here $\mathcal{C}(\boldsymbol{y}, \boldsymbol{f}) = \frac{1}{n} \sum_{i=0}^{n-1} (y_i - f_M(x_i))^2$ with $f_M(x) = \sum_{i=1}^{M} \beta_m b(x; \gamma_m)$.
2. Initialize with a guess $f_0(x)$. It could be one or even zero or some random numbers.
3. For $m = 1 : M$
   (a) minimize $\sum_{i=0}^{n-1} (y_i - f_{m-1}(x_i) - \beta b(x; \gamma))^2$ wrt γ and β

(b) This gives the optimal values β_m and γ_m

(c) Determine then the new values $f_m(x) = f_{m-1}(x) + \beta_m b(x;\gamma_m)$

We could use any of the algorithms we have discussed till now. If we use trees, γ parameterizes the split variables and split points at the internal nodes, and the predictions at the terminal nodes.

Squared-Error Example and Iterative Fitting

To better understand what happens, let us develop the steps for the iterative fitting using the above squared error function.

For simplicity we assume also that our functions $b(x;\gamma) = 1 + \gamma x$.

This means that for every iteration m, we need to optimize

$$(\beta_m, \gamma_m) = \text{argmin}_{\beta,\lambda} \sum_{i=0}^{n-1} (y_i - f_{m-1}(x_i) - \beta b(x;\gamma))^2 = \sum_{i=0}^{n-1} (y_i - f_{m-1}(x_i) - \beta(1+\gamma x_i))^2.$$

We start our iteration by simply setting $f_0(x) = 0$. Taking the derivatives with respect to β and γ we obtain

$$\frac{\partial \mathcal{C}}{\partial \beta} = -2 \sum_i (1 + \gamma x_i)(y_i - \beta(1 + \gamma x_i)) = 0,$$

and

$$\frac{\partial \mathcal{C}}{\partial \gamma} = -2 \sum_i \beta x_i (y_i - \beta(1 + \gamma x_i)) = 0.$$

We can then rewrite these equations as (defining $\boldsymbol{w} = \boldsymbol{e} + \gamma \boldsymbol{x}$) with $\boldsymbol{e}$ being the unit vector)

$$\gamma \boldsymbol{w}^T (\boldsymbol{y} - \beta \gamma \boldsymbol{w}) = 0,$$

which gives us $\beta = \boldsymbol{w}^T \boldsymbol{y}/(\boldsymbol{w}^T \boldsymbol{w})$. Similarly we have

$$\beta \gamma \boldsymbol{x}^T (\boldsymbol{y} - \beta(1 + \gamma \boldsymbol{x})) = 0,$$

which leads to $\gamma = (\boldsymbol{x}^T \boldsymbol{y} - \beta \boldsymbol{x}^T \boldsymbol{e})/(\beta \boldsymbol{x}^T \boldsymbol{x})$. Inserting for β gives us an equation for γ. This is a non-linear equation in the unknown γ and has to be solved numerically.

The solution to these two equations gives us in turn β_1 and γ_1 leading to the new expression for $f_1(x)$ as $f_1(x) = \beta_1(1 + \gamma_1 x)$. Doing this M times results in our final estimate for the function f.

Iterative Fitting, Classification and AdaBoost

Let us consider a binary classification problem with two outcomes $y_i \in \{-1, 1\}$ and $i = 0, 1, 2, \ldots, n-1$ as our set of observations. We define a classification function $G(x)$ which produces a prediction taking one or the other of the two values $\{-1, 1\}$.

The error rate of the training sample is then

$$\overline{\text{err}} = \frac{1}{n} \sum_{i=0}^{n-1} I(y_i \neq G(x_i)).$$

The iterative procedure starts with defining a weak classifier whose error rate is barely better than random guessing. The iterative procedure in boosting is to sequentially apply a weak classification algorithm to repeatedly modified versions of the data producing a sequence of weak classifiers $G_m(x)$.

Here we will express our function $f(x)$ in terms of $G(x)$. That is

$$f_M(x) = \sum_{i=1}^{M} \beta_m b(x; \gamma_m),$$

will be a function of

$$G_M(x) = \text{sign} \sum_{i=1}^{M} \alpha_m G_m(x).$$

Adaptive Boosting, AdaBoost

In our iterative procedure we define thus

$$f_m(x) = f_{m-1}(x) + \beta_m G_m(x).$$

The simplest possible cost function which leads (also simple from a computational point of view) to the AdaBoost algorithm is the exponential cost/loss function defined as

$$C(\boldsymbol{y}, \boldsymbol{f}) = \sum_{i=0}^{n-1} \exp\left(-y_i(f_{m-1}(x_i) + \beta G(x_i)\right).$$

We optimize β and G for each value of $m = 1 : M$ as we did in the regression case. This is normally done in two steps. Let us however first rewrite the cost function as

$$C(\boldsymbol{y}, \boldsymbol{f}) = \sum_{i=0}^{n-1} w_i^m \exp\left(-y_i \beta G(x_i)\right),$$

where we have defined $w_i^m = \exp\left(-y_i f_{m-1}(x_i)\right)$.

Building up AdaBoost

First, for any $\beta > 0$, we optimize G by setting

$$G_m(x) = \text{sign} \sum_{i=0}^{n-1} w_i^m I(y_i \neq G_(x_i)),$$

which is the classifier that minimizes the weighted error rate in predicting y.

We can do this by rewriting

$$\exp -(\beta) \sum_{y_i = G(x_i)} w_i^m + \exp(\beta) \sum_{y_i \neq G(x_i)} w_i^m,$$

which can be rewritten as

$$(\exp(\beta) - \exp -(\beta)) \sum_{i=0}^{n-1} w_i^m I(y_i \neq G(x_i)) + \exp(-\beta) \sum_{i=0}^{n-1} w_i^m = 0,$$

which leads to

$$\beta_m = \frac{1}{2} \log \frac{1 - \overline{\text{err}}}{\overline{\text{err}}},$$

where we have redefined the error as

$$\overline{\text{err}}_m = \frac{1}{n} \frac{\sum_{i=0}^{n-1} w_i^m I(y_i \neq G(x_i)}{\sum_{i=0}^{n-1} w_i^m},$$

which leads to an update of

$$f_m(x) = f_{m-1}(x) + \beta_m G_m(x).$$

This leads to the new weights

$$w_i^{m+1} = w_i^m \exp(-y_i \beta_m G_m(x_i))$$

Adaptive boosting: AdaBoost, Basic Algorithm

The algorithm here is rather straightforward. Assume that our weak classifier is a decision tree and we consider a binary set of outputs with $y_i \in \{-1, 1\}$ and $i = 0, 1, 2, \ldots, n-1$ as our set of observations. Our design matrix is given in terms of the feature/predictor vectors $\boldsymbol{X} = [\boldsymbol{x}_0 \boldsymbol{x}_1 x_{p-1}]$. Finally, we define also a classifier determined by our data via a function $G(x)$. This function tells us how well we are able to classify our outputs/targets $\boldsymbol{y}$.

We have already defined the misclassification error err as

$$\text{err} = \frac{1}{n} \sum_{i=0}^{n-1} I(y_i \neq G(x_i)),$$

where the function $I()$ is one if we misclassify and zero if we classify correctly.

Basic Steps of AdaBoost

With the above definitions we are now ready to set up the algorithm for AdaBoost. The basic idea is to set up weights which will be used to scale the correctly classified and the misclassified cases.

1. We start by initializing all weights to $w_i = 1/n$, with $i = 0, 1, 2, \ldots n-1$. It is easy to see that we must have $\sum_{i=0}^{n-1} w_i = 1$.

2. We rewrite the misclassification error as

$$\overline{\mathrm{err}}_m = \frac{\sum_{i=0}^{n-1} w_i^m I(y_i \neq G(x_i))}{\sum_{i=0}^{n-1} w_i},$$

1. Then we start looping over all attempts at classifying, namely we start an iterative process for $m = 1 : M$, where M is the final number of classifications. Our given classifier could for example be a plain decision tree.

   (a) Fit then a given classifier to the training set using the weights w_i.

   (b) Compute then err and figure out which events are classified properly and which are classified wrongly.

   (c) Define a quantity $\alpha_m = \log\left(1 - \overline{\mathrm{err}}_m\right)/\overline{\mathrm{err}}_m$

   (d) Set the new weights to $w_i = w_i \times \exp\left(\alpha_m I(y_i \neq G(x_i)\right.$.

2. Compute the new classifier $G(x) = \sum_{i=0}^{n-1} \alpha_m I(y_i \neq G(x_i)$.

For the iterations with $m \leq 2$ the weights are modified individually at each steps. The observations which were misclassified at iteration $m-1$ have a weight which is larger than those which were classified properly. As this proceeds, the observations which were difficult to classifiy correctly are given a larger influence. Each new classification step m is then forced to concentrate on those observations that are missed in the previous iterations.

AdaBoost Examples

Using **Scikit-Learn** it is easy to apply the adaptive boosting algorithm, as done here.

```
from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5, random_state=42)
ada_clf.fit(X_train, y_train)

from sklearn.ensemble import AdaBoostClassifier

ada_clf = AdaBoostClassifier(
    DecisionTreeClassifier(max_depth=1), n_estimators=200,
    algorithm="SAMME.R", learning_rate=0.5, random_state=42)
```

```
ada_clf.fit(X_train_scaled, y_train)
y_pred = ada_clf.predict(X_test_scaled)
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=True)
plt.show()
y_probas = ada_clf.predict_proba(X_test_scaled)
skplt.metrics.plot_roc(y_test, y_probas)
plt.show()
skplt.metrics.plot_cumulative_gain(y_test, y_probas)
plt.show()
```

Gradient boosting: Basics with Steepest Descent/Functional Gradient Descent

Gradient boosting is again a similar technique to Adaptive boosting, it combines so-called weak classifiers or regressors into a strong method via a series of iterations.

In order to understand the method, let us illustrate its basics by bringing back the essential steps in linear regression, where our cost function was the least squares function.

The Squared-Error again! Steepest Descent

We start again with our cost function $\mathcal{C}(\boldsymbol{y}m\boldsymbol{f}) = \sum_{i=0}^{n-1} \mathcal{L}(y_i, f(x_i))$ where we want to minimize This means that for every iteration, we need to optimize

$$(\hat{\boldsymbol{f}}) = \text{argmin}_{\boldsymbol{f}} \sum_{i=0}^{n-1} (y_i - f(x_i))^2.$$

We define a real function $h_m(x)$ that defines our final function $f_M(x)$ as

$$f_M(x) = \sum_{m=0}^{M} h_m(x).$$

In the steepest decent approach we approximate $h_m(x) = -\rho_m g_m(x)$, where ρ_m is a scalar and $g_m(x)$ the gradient defined as

$$g_m(x_i) = \left[\frac{\partial \mathcal{L}(y_i, f(x_i))}{\partial f(x_i)} \right]_{f(x_i)=f_{m-1}(x_i)}.$$

With the new gradient we can update $f_m(x) = f_{m-1}(x) - \rho_m g_m(x)$. Using the above squared-error function we see that the gradient is $g_m(x_i) = -2(y_i - f(x_i))$.

Choosing $f_0(x) = 0$ we obtain $g_m(x) = -2y_i$ and inserting this into the minimization problem for the cost function we have

$$(\rho_1) = \text{argmin}_{\rho} \sum_{i=0}^{n-1} (y_i + 2\rho y_i)^2.$$

Steepest Descent Example

Optimizing with respect to ρ we obtain (taking the derivative) that $\rho_1 = -1/2$. We have then that

$$f_1(x) = f_0(x) - \rho_1 g_1(x) = -y_i.$$

We can then proceed and compute

$$g_2(x_i) = \left[\frac{\partial \mathcal{L}(y_i, f(x_i))}{\partial f(x_i)}\right]_{f(x_i)=f_1(x_i)=y_i} = -4y_i,$$

and find a new value for $\rho_2 = -1/2$ and continue till we have reached $m = M$. We can modify the steepest descent method, or steepest boosting, by introducing what is called **gradient boosting**.

Gradient Boosting, algorithm

Steepest descent is however not much used, since it only optimizes f at a fixed set of n points, so we do not learn a function that can generalize. However, we can modify the algorithm by fitting a weak learner to approximate the negative gradient signal.

Suppose we have a cost function $C(f) = \sum_{i=0}^{n-1} L(y_i, f(x_i))$ where y_i is our target and $f(x_i)$ the function which is meant to model y_i. The above cost function could be our standard squared-error function

$$C(\boldsymbol{y}, \boldsymbol{f}) = \sum_{i=0}^{n-1}(y_i - f(x_i))^2.$$

The way we proceed in an iterative fashion is to

1. Initialize our estimate $f_0(x)$.
2. For $m = 1 : M$, we
   (a) compute the negative gradient vector $\boldsymbol{u}_m = -\partial C(\boldsymbol{y}, \boldsymbol{f})/\partial \boldsymbol{f}(x)$ at $f(x) = f_{m-1}(x)$;
   (b) fit the so-called base-learner to the negative gradient $h_m(u_m, x)$;
   (c) update the estimate $f_m(x) = f_{m-1}(x) + h_m(u_m, x)$;
3. The final estimate is then $f_M(x) = \sum_{m=1}^{M} h_m(u_m, x)$.

Gradient Boosting, Examples of Regression

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.preprocessing import StandardScaler
```

```
import scikitplot as skplt
from sklearn.metrics import mean_squared_error

n = 100
maxdegree = 6

# Make data set.
x = np.linspace(-3, 3, n).reshape(-1, 1)
y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2)+ np.random.normal(0, 0.1, x.shape)

error = np.zeros(maxdegree)
bias = np.zeros(maxdegree)
variance = np.zeros(maxdegree)
polydegree = np.zeros(maxdegree)
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

for degree in range(1,maxdegree):
    model = GradientBoostingRegressor(max_depth=degree, n_estimators=100, learning_rate=1.0)
    model.fit(X_train_scaled,y_train)
    y_pred = model.predict(X_test_scaled)
    polydegree[degree] = degree
    error[degree] = np.mean( np.mean((y_test - y_pred)**2) )
    bias[degree] = np.mean( (y_test - np.mean(y_pred))**2 )
    variance[degree] = np.mean( np.var(y_pred) )
    print('Max depth:', degree)
    print('Error:', error[degree])
    print('Bias^2:', bias[degree])
    print('Var:', variance[degree])
    print('{} >= {} + {} = {}'.format(error[degree], bias[degree], variance[degree], bias[degree]+

plt.xlim(1,maxdegree-1)
plt.plot(polydegree, error, label='Error')
plt.plot(polydegree, bias, label='bias')
plt.plot(polydegree, variance, label='Variance')
plt.legend()
save_fig("gdregression")
plt.show()
```

Gradient Boosting, Classification Example

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
import scikitplot as skplt
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import cross_validate

# Load the data
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_state=0)
print(X_train.shape)
print(X_test.shape)
#now scale the data
from sklearn.preprocessing import StandardScaler
```

```
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

gd_clf = GradientBoostingClassifier(max_depth=3, n_estimators=100, learning_rate=1.0)
gd_clf.fit(X_train_scaled, y_train)
#Cross validation
accuracy = cross_validate(gd_clf,X_test_scaled,y_test,cv=10)['test_score']
print(accuracy)
print("Test set accuracy with Random Forests and scaled data: {:.2f}".format(gd_clf.score(X_test_s

import scikitplot as skplt
y_pred = gd_clf.predict(X_test_scaled)
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=True)
save_fig("gdclassiffierconfusion")
plt.show()
y_probas = gd_clf.predict_proba(X_test_scaled)
skplt.metrics.plot_roc(y_test, y_probas)
save_fig("gdclassiffierroc")
plt.show()
skplt.metrics.plot_cumulative_gain(y_test, y_probas)
save_fig("gdclassiffiercgain")
plt.show()
```

XGBoost: Extreme Gradient Boosting

XGBoost or Extreme Gradient Boosting, is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It implements machine learning algorithms under the Gradient Boosting framework. XGBoost provides a parallel tree boosting that solve many data science problems in a fast and accurate way. See the article by Chen and Guestrin.

The authors design and build a highly scalable end-to-end tree boosting system. It has a theoretically justified weighted quantile sketch for efficient proposal calculation. It introduces a novel sparsity-aware algorithm for parallel tree learning and an effective cache-aware block structure for out-of-core tree learning.

It is now the algorithm which wins essentially all ML competitions!!!

Regression Case

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import train_test_split
import xgboost as xgb
from sklearn.preprocessing import StandardScaler
import scikitplot as skplt
from sklearn.metrics import mean_squared_error

n = 100
maxdegree = 6

# Make data set.
x = np.linspace(-3, 3, n).reshape(-1, 1)
y = np.exp(-x**2) + 1.5 * np.exp(-(x-2)**2)+ np.random.normal(0, 0.1, x.shape)
```

```
error = np.zeros(maxdegree)
bias = np.zeros(maxdegree)
variance = np.zeros(maxdegree)
polydegree = np.zeros(maxdegree)
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.2)
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)

for degree in range(maxdegree):
    model =  xgb.XGBRegressor(objective ='reg:squarederror', colsaobjective ='reg:squarederror', c

    model.fit(X_train_scaled,y_train)
    y_pred = model.predict(X_test_scaled)
    polydegree[degree] = degree
    error[degree] = np.mean( np.mean((y_test - y_pred)**2) )
    bias[degree] = np.mean( (y_test - np.mean(y_pred))**2 )
    variance[degree] = np.mean( np.var(y_pred) )
    print('Max depth:', degree)
    print('Error:', error[degree])
    print('Bias^2:', bias[degree])
    print('Var:', variance[degree])
    print('{} >= {} + {} = {}'.format(error[degree], bias[degree], variance[degree], bias[degree]+

plt.xlim(1,maxdegree-1)
plt.plot(polydegree, error, label='Error')
plt.plot(polydegree, bias, label='bias')
plt.plot(polydegree, variance, label='Variance')
plt.legend()
plt.show()
```

Xgboost on the Cancer Data

As you will see from the confusion matrix below, XGBoots does an excellent job on the Wisconsin cancer data and outperforms essentially all agorithms we have discussed till now.

```
import matplotlib.pyplot as plt
import numpy as np
from sklearn.model_selection import  train_test_split
from sklearn.datasets import load_breast_cancer
from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import cross_validate
import scikitplot as skplt
import xgboost as xgb
# Load the data
cancer = load_breast_cancer()

X_train, X_test, y_train, y_test = train_test_split(cancer.data,cancer.target,random_state=0)
print(X_train.shape)
print(X_test.shape)
#now scale the data
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X_train)
X_train_scaled = scaler.transform(X_train)
X_test_scaled = scaler.transform(X_test)
```

```
xg_clf = xgb.XGBClassifier()
xg_clf.fit(X_train_scaled,y_train)

y_test = xg_clf.predict(X_test_scaled)

print("Test set accuracy with Random Forests and scaled data: {:.2f}".format(xg_clf.score(X_test_s

import scikitplot as skplt
y_pred = xg_clf.predict(X_test_scaled)
skplt.metrics.plot_confusion_matrix(y_test, y_pred, normalize=True)
save_fig("xdclassiffierconfusion")
plt.show()
y_probas = xg_clf.predict_proba(X_test_scaled)
skplt.metrics.plot_roc(y_test, y_probas)
save_fig("xdclassiffierroc")
plt.show()
skplt.metrics.plot_cumulative_gain(y_test, y_probas)
save_fig("gdclassiffiercgain")
plt.show()


xgb.plot_tree(xg_clf,num_trees=0)
plt.rcParams['figure.figsize'] = [50, 10]
save_fig("xgtree")
plt.show()

xgb.plot_importance(xg_clf)
plt.rcParams['figure.figsize'] = [5, 5]
save_fig("xgparams")
plt.show()
```

Topics we have covered

The course has two central parts

1. Statistical analysis and optimization of data
2. Machine learning

Statistical analysis and optimization of data

The following topics have been discussed:

1. Basic concepts, expectation values, variance, covariance, correlation functions and errors;
2. Gradient methods for data optimization
3. Estimation of errors using cross-validation, bootstrapping and jackknife methods;

Machine learning

The following topics will be covered

1. Linear methods for regression and classification:
   (a) Ordinary Least Squares
   (b) Ridge regression
   (c) Lasso regression
   (d) Logistic regression
2. Neural networks and deep learning:
   (a) Feed Forward Neural Networks
   (b) Convolutional Neural Networks
   (c) Recurrent Neural Networks
3. Decisions trees and ensemble methods:
   (a) Decision trees
   (b) Bagging and voting
   (c) Random forests
   (d) Boosting and gradient boosting

Learning outcomes and overarching aims of this course

The course introduces a variety of central algorithms and methods essential for studies of data analysis and machine learning. The course is project based and through the various projects, normally three, you will be exposed to fundamental research problems in these fields, with the aim to reproduce state of the art scientific results. The students will learn to develop and structure large codes for studying these systems, get acquainted with computing facilities and learn to handle large scientific projects. A good scientific and ethical conduct is emphasized throughout the course.

- Understand linear methods for regression and classification;
- Learn about neural network;
- Learn about bagging, boosting and trees
- Learn about basic data analysis;
- Be capable of extending the acquired knowledge to other systems and cases;

- Have an understanding of central algorithms used in data analysis and machine learning;
- Work on numerical projects to illustrate the theory. The projects play a central role and you are expected to know modern programming languages like Python or C++.

Perspective on Machine Learning

1. Rapidly emerging application area
2. Experiment AND theory are evolving in many many fields. Still many low-hanging fruits.
3. Requires education/retraining for more widespread adoption
4. A lot of "word-of-mouth" development methods

Huge amounts of data sets require automation, classical analysis tools often inadequate. High energy physics hit this wall in the 90's. In 2009 single top quark production was determined via Boosted decision trees, Bayesian Neural Networks, etc.

Machine Learning Research

Where to find recent results:

1. Conference proceedings, arXiv and blog posts!
2. **NIPS**: Neural Information Processing Systems
3. **ICLR**: International Conference on Learning Representations
4. **ICML**: International Conference on Machine Learning
5. Journal of Machine Learning Research
6. Follow ML on ArXiv

Starting your Machine Learning Project

1. Identify problem type: classification, regression
2. Consider your data carefully
3. Choose a simple model that fits 1. and 2.
4. Consider your data carefully again! Think of data representation more carefully.
5. Based on your results, feedback loop to earliest possible point

Choose a Model and Algorithm

1. Supervised?
2. Start with the simplest model that fits your problem
3. Start with minimal processing of data

Preparing Your Data

1. Shuffle your data
2. Mean center your data
    - Why?
3. Normalize the variance
    - Why?
4. **Whitening**
    - Decorrelates data
    - Can be hit or miss
5. When to do train/test split?

Which Activation and Weights to Choose in Neural Networks

1. RELU? ELU?
2. Sigmoid or Tanh?
3. Set all weights to 0?
    - Terrible idea
4. Set all weights to random values?
    - Small random values

Optimization Methods and Hyperparameters

1. Stochastic gradient descent
   (a) Stochastic gradient descent + momentum
2. State-of-the-art approaches:
   - RMSProp
   - Adam
   - and more

Which regularization and hyperparameters? L_1 or L_2, soft classifiers, depths of trees and many other. Need to explore a large set of hyperparameters and regularization methods.

Resampling

When do we resample?

1. Bootstrap
2. Cross-validation
3. Jackknife and many other

What's the future like?

Based on multi-layer nonlinear neural networks, deep learning can learn directly from raw data, automatically extract and abstract features from layer to layer, and then achieve the goal of regression, classification, or ranking. Deep learning has made breakthroughs in computer vision, speech processing and natural language, and reached or even surpassed human level. The success of deep learning is mainly due to the three factors: big data, big model, and big computing.

In the past few decades, many different architectures of deep neural networks have been proposed, such as

1. Convolutional neural networks, which are mostly used in image and video data processing, and have also been applied to sequential data such as text processing;
2. Recurrent neural networks, which can process sequential data of variable length and have been widely used in natural language understanding and speech processing;
3. Encoder-decoder framework, which is mostly used for image or sequence generation, such as machine translation, text summarization, and image captioning.

Types of Machine Learning, a repetition

The approaches to machine learning are many, but are often split into two main categories. In *supervised learning* we know the answer to a problem, and let the computer deduce the logic behind it. On the other hand, *unsupervised learning* is a method for finding patterns and relationship in data sets without any prior knowledge of the system. Some authours also operate with a third category, namely *reinforcement learning*. This is a paradigm of learning inspired by behavioural psychology, where learning is achieved by trial-and-error, solely from rewards and punishment.

Another way to categorize machine learning tasks is to consider the desired output of a system. Some of the most common tasks are:

- Classification: Outputs are divided into two or more classes. The goal is to produce a model that assigns inputs into one of these classes. An example is to identify digits based on pictures of hand-written ones. Classification is typically supervised learning.
- Regression: Finding a functional relationship between an input data set and a reference data set. The goal is to construct a function that maps input data to continuous output values.
- Clustering: Data are divided into groups with certain common traits, without knowing the different groups beforehand. It is thus a form of unsupervised learning.
- Other unsupervised learning algortihms like **Boltzmann machines**

Autoencoders: Overarching view

Autoencoders are artificial neural networks capable of learning efficient representations of the input data (these representations are called codings) without any supervision (i.e., the training set is unlabeled). These codings typically have a much lower dimensionality than the input data, making autoencoders useful for dimensionality reduction.

More importantly, autoencoders act as powerful feature detectors, and they can be used for unsupervised pretraining of deep neural networks.

Lastly, they are capable of randomly generating new data that looks very similar to the training data; this is called a generative model. For example, you could train an autoencoder on pictures of faces, and it would then be able to generate new faces. Surprisingly, autoencoders work by simply learning to copy their inputs to their outputs. This may sound like a trivial task, but we will see that constraining the network in various ways can make it rather difficult. For example, you can limit the size of the internal representation, or you can add noise to the inputs and train the network to recover the original inputs. These constraints prevent the autoencoder from trivially copying the inputs directly to the outputs, which forces it to learn efficient ways of representing the data.

In short, the codings are byproducts of the autoencoder's attempt to learn the identity function under some constraints.

Video on autoencoders

See also A. Geron's textbook, chapter 15.

Bayesian Machine Learning

This is an important topic if we aim at extracting a probability distribution. This gives us also a confidence interval and error estimates.

Bayesian machine learning allows us to encode our prior beliefs about what those models should look like, independent of what the data tells us. This is especially useful when we don't have a ton of data to confidently learn our model.

Video on Bayesian deep learning

See also the slides here.

Reinforcement Learning

Reinforcement Learning (RL) is one of the most exciting fields of Machine Learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years.

It studies how agents take actions based on trial and error, so as to maximize some notion of cumulative reward in a dynamic system or environment. Due to its generality, the problem has also been studied in many other disciplines, such as game theory, control theory, operations research, information theory, multi-agent systems, swarm intelligence, statistics, and genetic algorithms.

In March 2016, AlphaGo, a computer program that plays the board game Go, beat Lee Sedol in a five-game match. This was the first time a computer Go program had beaten a 9-dan (highest rank) professional without handicaps. AlphaGo is based on deep convolutional neural networks and reinforcement learning. AlphaGo's victory was a major milestone in artificial intelligence and it has also made reinforcement learning a hot research area in the field of machine learning.

Lecture on Reinforcement Learning.

See also A. Geron's textbook, chapter 16.

Transfer learning

The goal of transfer learning is to transfer the model or knowledge obtained from a source task to the target task, in order to resolve the issues of insufficient training data in the target task. The rationality of doing so lies in that usually the source and target tasks have inter-correlations, and therefore either the features, samples, or models in the source task might provide useful information for us to better solve the target task. Transfer learning is a hot research topic in recent years, with many problems still waiting to be studied.

Lecture on transfer learning.

Adversarial learning

The conventional deep generative model has a potential problem: the model tends to generate extreme instances to maximize the probabilistic likelihood, which will hurt its performance. Adversarial learning utilizes the adversarial behaviors (e.g., generating adversarial instances or training an adversarial model) to enhance the robustness of the model and improve the quality of the generated data. In recent years, one of the most promising unsupervised learning technologies, generative adversarial networks (GAN), has already been successfully applied to image, speech, and text.

Lecture on adversial learning.

Dual learning

Dual learning is a new learning paradigm, the basic idea of which is to use the primal-dual structure between machine learning tasks to obtain effective feedback/regularization, and guide and strengthen the learning process, thus reducing the requirement of large-scale labeled data for deep learning. The idea of dual learning has been applied to many problems in machine learning, including machine translation, image style conversion, question answering and generation, image classification and generation, text classification and generation, image-to-text, and text-to-image.

Distributed machine learning

Distributed computation will speed up machine learning algorithms, significantly improve their efficiency, and thus enlarge their application. When distributed meets machine learning, more than just implementing the machine learning algorithms in parallel is required.

Meta learning

Meta learning is an emerging research direction in machine learning. Roughly speaking, meta learning concerns learning how to learn, and focuses on the understanding and adaptation of the learning itself, instead of just completing a specific learning task. That is, a meta learner needs to be able to evaluate its own learning methods and adjust its own learning methods according to specific learning tasks.

The Challenges Facing Machine Learning

While there has been much progress in machine learning, there are also challenges.

For example, the mainstream machine learning technologies are black-box approaches, making us concerned about their potential risks. To tackle this challenge, we may want to make machine learning more explainable and controllable. As another example, the computational complexity of machine learning algorithms is usually very high and we may want to invent lightweight algorithms

or implementations. Furthermore, in many domains such as physics, chemistry, biology, and social sciences, people usually seek elegantly simple equations (e.g., the Schrödinger equation) to uncover the underlying laws behind various phenomena. In the field of machine learning, can we reveal simple laws instead of designing more complex models for data fitting? Although there are many challenges, we are still very optimistic about the future of machine learning. As we look forward to the future, here are what we think the research hotspots in the next ten years will be.

See the article on Discovery of Physics From Data: Universal Laws and Discrepancies

Explainable machine learning

Machine learning, especially deep learning, evolves rapidly. The ability gap between machine and human on many complex cognitive tasks becomes narrower and narrower. However, we are still in the very early stage in terms of explaining why those effective models work and how they work.

What is missing: the gap between correlation and causation. Standard Machine Learning is based on what e have called a frequentist approach.

Most machine learning techniques, especially the statistical ones, depend highly on correlations in data sets to make predictions and analyses. In contrast, rational humans tend to reply on clear and trustworthy causality relations obtained via logical reasoning on real and clear facts. It is one of the core goals of explainable machine learning to transition from solving problems by data correlation to solving problems by logical reasoning.

Bayesian Machine Learning is one of the exciting research directions in this field.

Quantum machine learning

Quantum machine learning is an emerging interdisciplinary research area at the intersection of quantum computing and machine learning.

Quantum computers use effects such as quantum coherence and quantum entanglement to process information, which is fundamentally different from classical computers. Quantum algorithms have surpassed the best classical algorithms in several problems (e.g., searching for an unsorted database, inverting a sparse matrix), which we call quantum acceleration.

When quantum computing meets machine learning, it can be a mutually beneficial and reinforcing process, as it allows us to take advantage of quantum computing to improve the performance of classical machine learning algorithms. In addition, we can also use the machine learning algorithms (on classic computers) to analyze and improve quantum computing systems.

Lecture on Quantum ML.

Read interview with Maria Schuld on her work on Quantum Machine Learning. See also her recent textbook.

Quantum machine learning algorithms based on linear algebra

Many quantum machine learning algorithms are based on variants of quantum algorithms for solving linear equations, which can efficiently solve N-variable linear equations with complexity of O(log2 N) under certain conditions. The quantum matrix inversion algorithm can accelerate many machine learning methods, such as least square linear regression, least square version of support vector machine, Gaussian process, and more. The training of these algorithms can be simplified to solve linear equations. The key bottleneck of this type of quantum machine learning algorithms is data input—that is, how to initialize the quantum system with the entire data set. Although efficient data-input algorithms exist for certain situations, how to efficiently input data into a quantum system is as yet unknown for most cases.

Quantum reinforcement learning

In quantum reinforcement learning, a quantum agent interacts with the classical environment to obtain rewards from the environment, so as to adjust and improve its behavioral strategies. In some cases, it achieves quantum acceleration by the quantum processing capabilities of the agent or the possibility of exploring the environment through quantum superposition. Such algorithms have been proposed in superconducting circuits and systems of trapped ions.

Quantum deep learning

Dedicated quantum information processors, such as quantum annealers and programmable photonic circuits, are well suited for building deep quantum networks. The simplest deep quantum network is the Boltzmann machine. The classical Boltzmann machine consists of bits with tunable interactions and is trained by adjusting the interaction of these bits so that the distribution of its expression conforms to the statistics of the data. To quantize the Boltzmann machine, the neural network can simply be represented as a set of interacting quantum spins that correspond to an adjustable Ising model. Then, by initializing the input neurons in the Boltzmann machine to a fixed state and allowing the system to heat up, we can read out the output qubits to get the result.

Social machine learning

Machine learning aims to imitate how humans learn. While we have developed successful machine learning algorithms, until now we have ignored one important fact: humans are social. Each of us is one part of the total society and it is difficult for us to live, learn, and improve ourselves, alone and isolated. Therefore, we should design machines with social properties. Can we let machines evolve by imitating human society so as to achieve more effective, intelligent, interpretable "social machine learning"?

And much more.

The last words?

Early computer scientist Alan Kay said, **The best way to predict the future is to create it**. Therefore, all machine learning practitioners, whether scholars or engineers, professors or students, need to work together to advance these important research topics. Together, we will not just predict the future, but create it.