Trees and Forests

Professor Ernesto Lee

What we will cover

- Decision trees and the decision tree learning algorithm
- Ensambles
- Random forest: putting multiple trees together into one model
- Gradient boosting as an alternative way of combining decision trees

The Project

How do we apply CREDIT RISK if you work at a Bank?

1. First, we get the data and do some initial pre-processing.
2. Next, we train a decision tree model from Scikit-Learn for predicting the probability of default.
3. After that, we explain how decision trees work, which parameters the model has, and show how to adjust these parameters to get the best performance.
4. Then we combine multiple decision trees into one model – random forest. We look at its parameters and tune them to achieve the best predictive performance.
5. Finally, we explore a different way of combining decision trees – gradient boosting. We use XGBoost – a highly efficient library that implements gradient boosting. We'll train a model and tune its parameters.

The Data

https://bit.ly/MLTrain

Or

```
!wget
https://raw.githubusercontent.com/fenago
/pythonml/main/data/CreditScoring.csv
```

1 Status	credit status
2 Seniority	job seniority (years)
3 Home	type of home ownership
4 Time	time of requested loan
5 Age	client's age
6 Marital	marital status
7 Records	existance of records
8 Job	type of job
9 Expenses	amount of expenses
10 Income	amount of income
11 Assets	amount of assets
12 Debt	amount of debt
13 Amount	amount requested of loan
14 Price	price of good

Do Your Standard Imports

```
import pandas as pd

import numpy as np

import seaborn as sns

from matplotlib import pyplot as plt

%matplotlib inline

df = pd.read_csv('./data/CreditScoring.csv')
```

Data Cleaning

```
df.head()

df.columns = df.columns.str.lower()
```

Categorical Treatment

Find Unique Values for Categorical Columns

```
n = df.nunique(axis=0)

print("No.of.unique values in each column :\n", n)



# n = len(pd.unique(df[‘status’]))

# print("No.of.unique values :", n)
```

Categorical Data Treatment

Status – whether the customer managed to pay back the loan (1) or not (2)

Seniority – job experience in years

Home – the type of homeownership: renting (1), a homeowner (2), and others.

Time – period planned for the loan (in months).

Age – the age of the client.

Marital [status] – single (1), married (2), and others.

Records – whether the client has any previous records: no (1), yes (2).

Job – the type of job: full-time (1), part-time (2), and others.

Expenses – how much the client spends per month.

Income – how much the client earns per month.

Assets – the total worth of all the assets of the client.

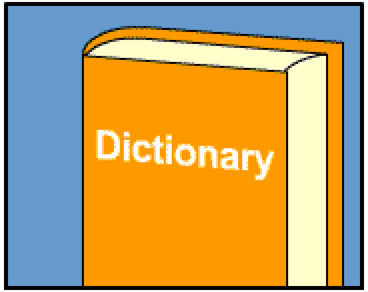
Debt – the amount of credit debt.

Amount – the requested amount of the loan.

Price – the price of an item the client wants to buy.

Dictionary Maps

```
status_values = {
    1: 'ok',
    2: 'default',
    0: 'unk'
}
df.status = df.status.map(status_values)
```



Exercise - 10 minutes

You do the same for the other categorical columns (home, marital status, record, job). Here is an example:

```
home_values = {
    1: 'rent',
    2: 'owner',
    3: 'private',
    4: 'ignore',
    5: 'parents',
    6: 'other',
    0: 'unk'
}
df.home = df.home.map(home_values)
```

Numerical Treatment

Summary Statistics

```
df.describe().round()
```

[14]:

	seniority	time	age	expenses	income	assets	debt	amount	price
count	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0	4455.0
mean	8.0	46.0	37.0	56.0	763317.0	1060341.0	404382.0	1039.0	1463.0
std	8.0	15.0	11.0	20.0	8703625.0	10217569.0	6344253.0	475.0	628.0
min	0.0	6.0	18.0	35.0	0.0	0.0	0.0	100.0	105.0
25%	2.0	36.0	28.0	35.0	80.0	0.0	0.0	700.0	1118.0
50%	5.0	48.0	36.0	51.0	120.0	3500.0	0.0	1000.0	1400.0
75%	12.0	60.0	45.0	72.0	166.0	6000.0	0.0	1300.0	1692.0
max	48.0	72.0	68.0	180.0	99999999.0	99999999.0	99999999.0	5000.0	11140.0

Encode Missing Numbers Properly

```
for c in ['income', 'assets', 'debt']:
    df[c] = df[c].replace(to_replace=99999999, value=np.nan)
```

Understand the Target Variable Balance

```
df.status.value_counts()
```

```
df = df[df.status != 'unk']
```

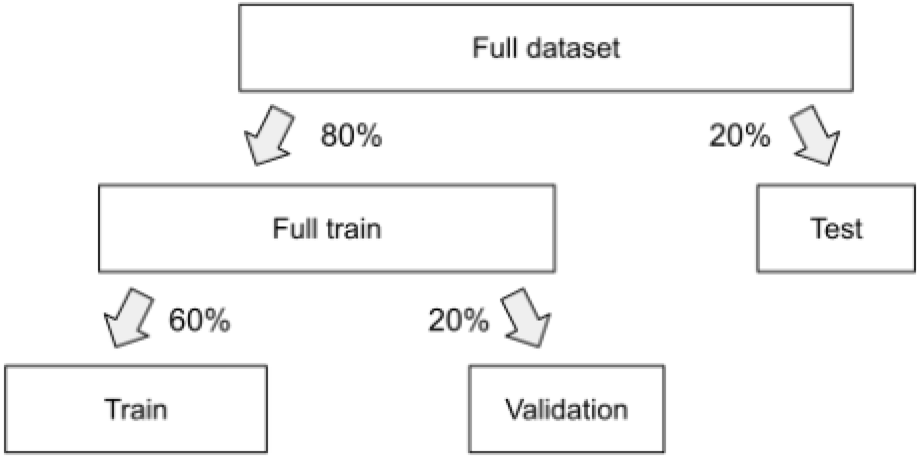
Dataset Preperation

What we will cover

- Split the dataset into train, validation, and test
- Handle missing values
- Use one-hot encoding to encode categorical variables
- Create the feature matrix X and the target variable y

Split Your Data

- Training data (60% of the original dataset)
- Validation data (20%)
- Test data (20%)



Split Your data

```
from sklearn.model_selection import train_test_split

df_train_full, df_test = train_test_split(df, test_size=0.2,
random_state=11)

df_train, df_val = train_test_split(df_train_full,
test_size=0.25, random_state=11)

len(df_train), len(df_val), len(df_test)
```

Separating the Target Variable

```
y_train = (df_train.status == 'default').values
y_val = (df_val.status == 'default').values


del df_train['status']
del df_val['status']
```

Replace Missing Values with a ZERO

```
df_train = df_train.fillna(0)

df_val = df_val.fillna(0)
```

One Hot Encode Your Categorical Data - Part I

```
# Dict Vectorizer needs a list of Dictionaries...

dict_train = df_train.to_dict(orient='records')

dict_val = df_val.to_dict(orient='records')
```

Each Dictionary in the Result Now Looks Like This

```
{'seniority': 10,
 'home': 'owner',
 'time': 36,
 'age': 36,
 'marital': 'married',
 'records': 'no',
 'job': 'freelance',
 'expenses': 75,
 'income': 0.0,
 'assets': 10000.0,
 'debt': 0.0,
 'amount': 1000,
 'price': 1400}
```

Use the List of Dictionaries As Input Into the DictVectorizer

```
from sklearn.feature_extraction import DictVectorizer

dv = DictVectorizer(sparse=False)

X_train = dv.fit_transform(dict_train)
X_val = dv.transform(dict_val)
```

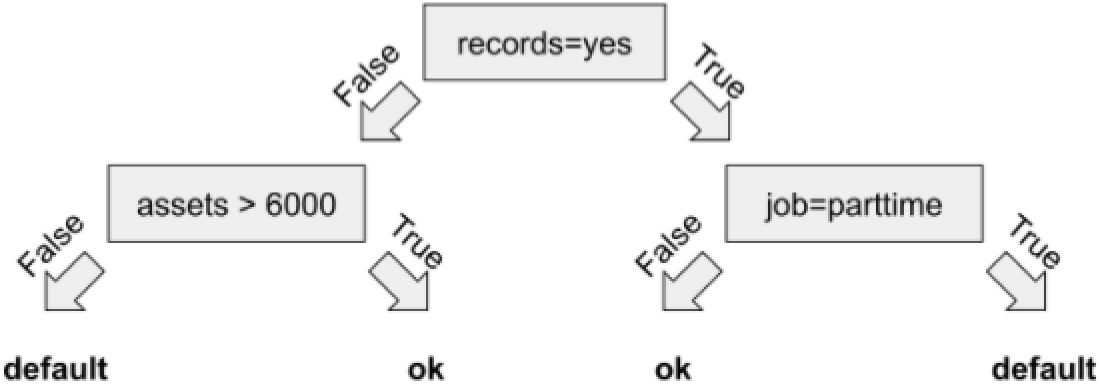
We are Now Ready to Train A Model!!

1. Do your imports
2. Make all column headers lowercase
3. Apply treatments to categorical data (unique values, dictionary maps)
4. Numerical treatments (descriptive statistics) - encode missing numbers with np.nan
5. Understand your target variables balance
6. Split your data: X_train, X_val, Separate your target variable y_train, y_val - replace missing values with a 0
7. One hot encode your categorical data by creating a list of dictionaries then using the DictVectorizer

Decision Trees

What is a Decision Tree

```
def assess_risk(client):
    if client['records'] == 'yes':
        if client['job'] == 'parttime':
            return 'default'
        else:
            return 'ok'
    else:
        if client['assets'] > 6000:
            return 'ok'
        else:
            return 'default'
```



Use Scikit Learn to do the same thing

```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier()

dt.fit(X_train, y_train)
```

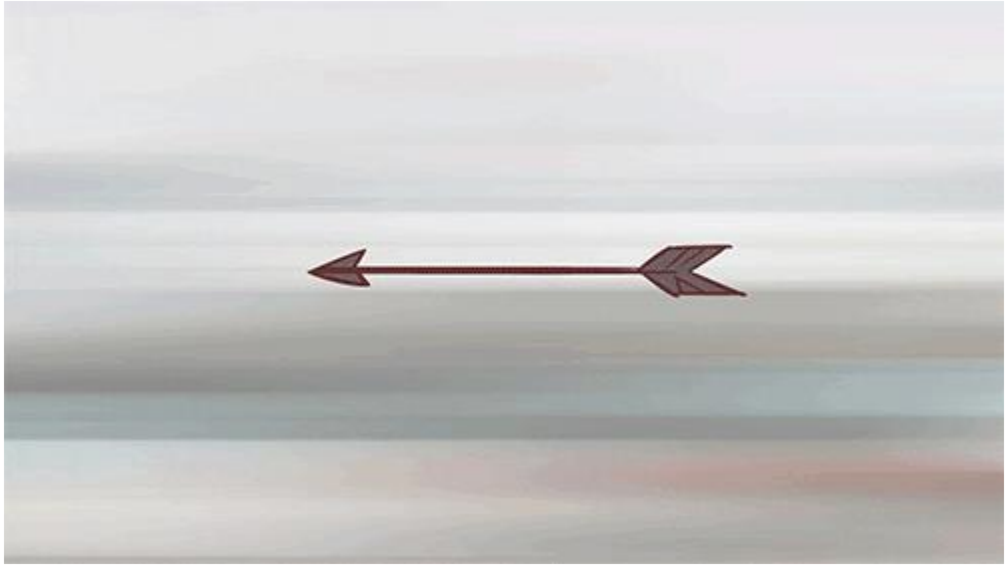
```python
# Compare Algorithms
from pandas import read_csv
from matplotlib import pyplot
from sklearn.model_selection import KFold
from sklearn.model_selection import cross_val_score
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.naive_bayes import GaussianNB
from sklearn.svm import SVC
models = []
models.append(('LR', LogisticRegression(solver='liblinear')))
models.append(('LDA', LinearDiscriminantAnalysis()))
models.append(('KNN', KNeighborsClassifier()))
models.append(('CART', DecisionTreeClassifier()))
models.append(('NB', GaussianNB()))
models.append(('SVM', SVC(gamma='auto')))
# evaluate each model in turn
results = []
names = []
scoring = 'accuracy'
for name, model in models:
    kfold = KFold(n_splits=10, random_state=7, shuffle=True)
    cv_results = cross_val_score(model, X_train, y_train, cv=kfold, scoring=scoring)
    results.append(cv_results)
    names.append(name)
    msg = "%s: %f (%f)" % (name, cv_results.mean(), cv_results.std())
    print(msg)
# boxplot algorithm comparison
fig = pyplot.figure()
fig.suptitle('Algorithm Comparison')
ax = fig.add_subplot(111)
pyplot.boxplot(results)
ax.set_xticklabels(names)
pyplot.show()
```

Is this a Good Model?

```
from sklearn.metrics import roc_auc_score

y_pred = dt.predict_proba(X_train)[:, 1]

roc_auc_score(y_train, y_pred)

y_pred = dt.predict_proba(X_val)[:, 1]

roc_auc_score(y_val, y_pred)
```

The Accuracy is 65%

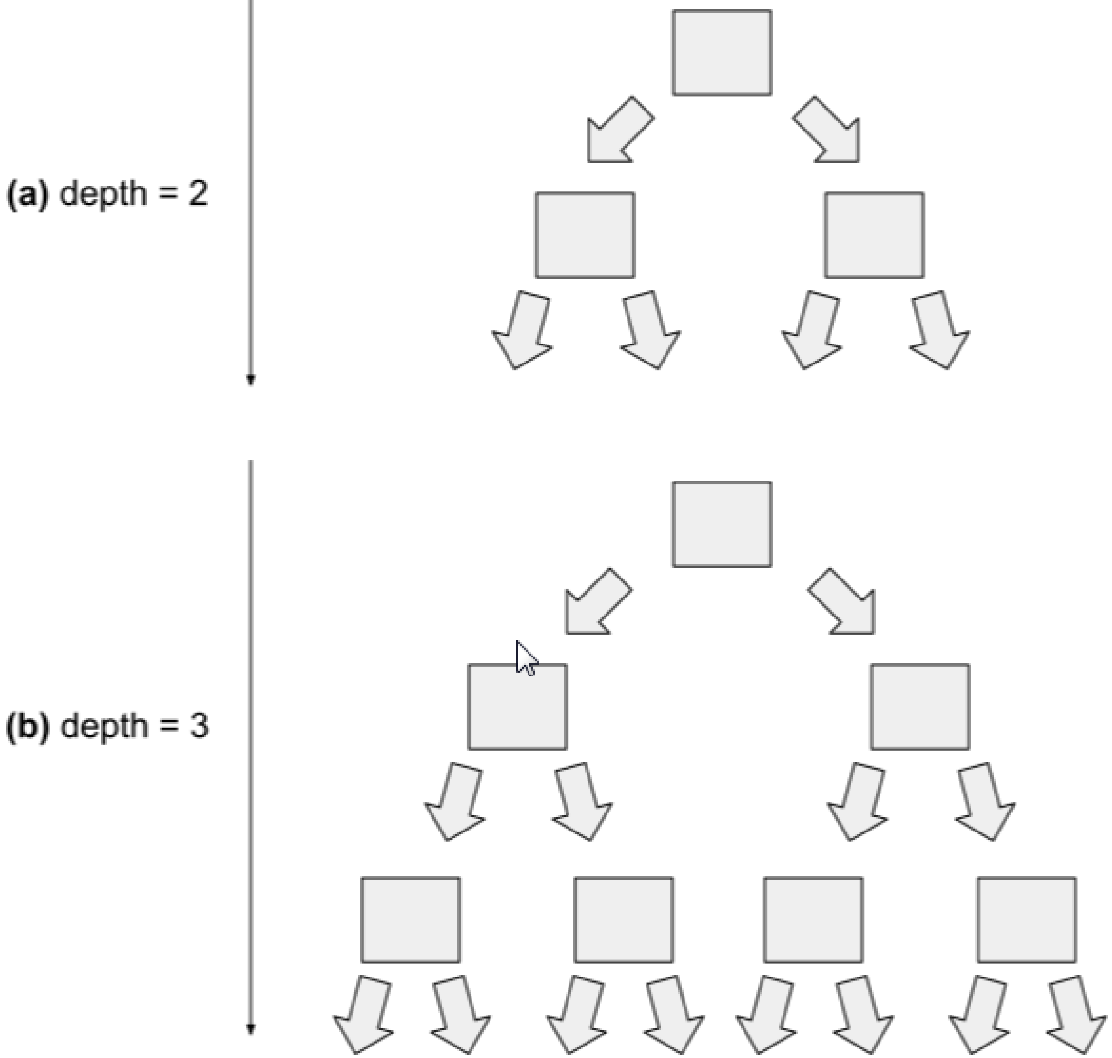
This is only a little better than a 50/50 guess.



The Tree Is Overfit

A tree with more levels can learn more complex rules.

A tree with two less levels is less complex than a tree with 3 and LESS prone to overfitting.

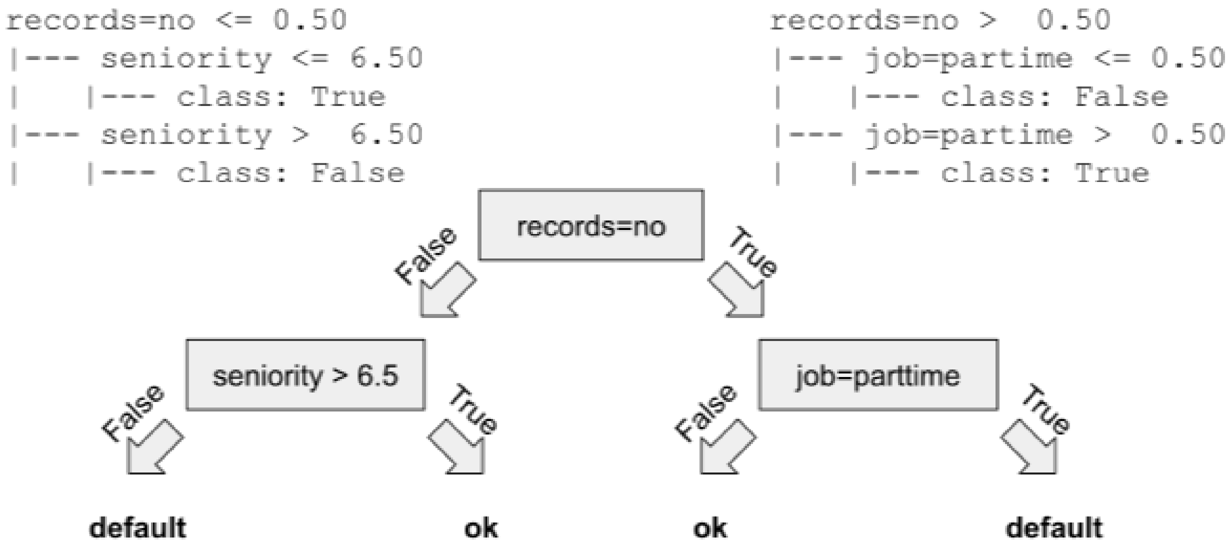


Reduce the Max Depth to Fight Overfitting

```
dt = DecisionTreeClassifier(max_depth=2)
dt.fit(X_train, y_train)
from sklearn.tree import export_text
tree_text = export_text(dt, feature_names=dv.feature_names_)
print(tree_text)
```

Features in the Tree

```
|--- records=no <= 0.50
|   |--- seniority <= 6.50
|   |   |--- class: True
|   |--- seniority >  6.50
|   |   |--- class: False
|--- records=no >  0.50
|   |--- job=partime <= 0.50
|   |   |--- class: False
|   |--- job=partime >  0.50
|   |   |--- class: True
```



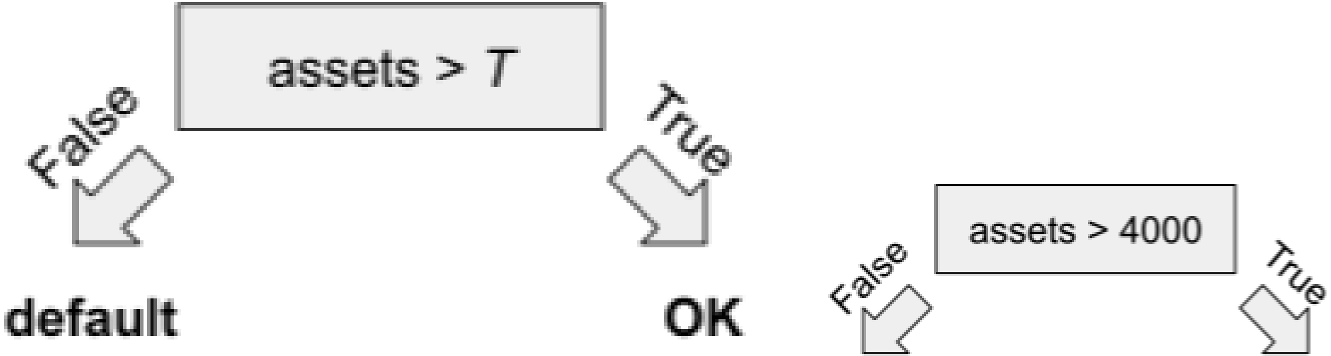
Retrain and Check the Accuracy

```
y_pred = dt.predict_proba(X_train)[:, 1]
auc = roc_auc_score(y_train, y_pred)
print('train auc', auc)

y_pred = dt.predict_proba(X_val)[:, 1]
auc = roc_auc_score(y_val, y_pred)
print('validation auc', auc)
```

How Do Decision Trees Learn?

Let's USe a Smaller Dataset

	assets	status
0	8000	default
1	2000	OK
2	0	OK
3	6000	OK
4	6000	default
5	9000	default



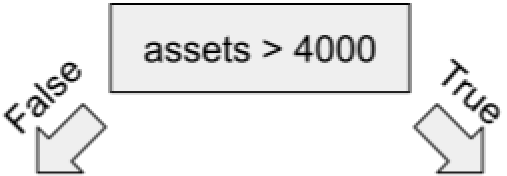
	assets	status
1	2000	default
2	0	default
5	4000	OK
7	3000	default

	assets	status
0	8000	default
3	5000	OK
4	5000	OK
6	9000	OK

Scikit-Learn View

```
from sklearn import tree

X = [[2000], [0],[4000],[3000],[8000],[5000],[5000],[9000]]

Y = [1,1,0,1,1,0,0,0]

clf = tree.DecisionTreeClassifier()

clf = clf.fit(X, Y)

clf.predict([[1001]])
```

How Trees Work

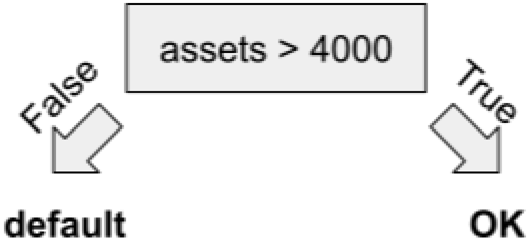


3 default
1 OK

	assets	status
1	2000	default
2	0	default
5	4000	OK
7	3000	default

	assets	status
0	8000	default
3	5000	OK
4	5000	OK
6	9000	OK

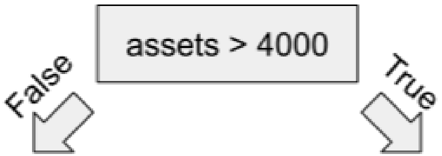
1 default
3 OK



Impurity

These groups should be as homogeneous as possible. Ideally, each group should contain only observations of one class.

- Try all possible values of T
- For each T, split the dataset into left and right groups and measure their impurity
- Select T that has the lowest degree of impurity



3 default
1 OK

4 total

	assets	status
1	2000	default
2	0	default
5	4000	OK
7	3000	default

impurity: 1/4

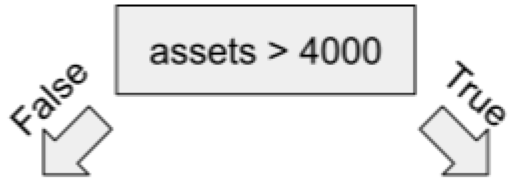
	assets	status
0	8000	default
3	5000	OK
4	5000	OK
6	9000	OK

impurity: 1/4

1 default
3 OK

4 total

Compute the Impurity



3 default
1 OK

4 total

	assets	status
1	2000	default
2	0	default
5	4000	OK
7	3000	default

impurity: 1/4

	assets	status
0	8000	default
3	5000	OK
4	5000	OK
6	9000	OK

1 default
3 OK

4 total

impurity: 1/4

assets > 2000

False

	assets	status
1	2000	default
2	0	default

True

	assets	status
0	8000	default
3	5000	OK
4	5000	OK
5	4000	OK
6	9000	OK
7	3000	default

assets > 3000

False

	assets	status
1	2000	default
2	0	default
7	3000	default

True

	assets	status
0	8000	default
3	5000	OK
4	5000	OK
5	4000	OK
6	9000	OK

assets > 5000

False

	assets	status
1	2000	default
2	0	default
3	5000	OK
4	5000	OK
5	4000	OK
7	3000	default

True

	assets	status
0	8000	default
6	9000	OK

Select the Best Feature For Splitting

	assets	debt	status
0	8000	3000	default
1	2000	1000	default
2	0	1000	default
3	5000	1000	OK
4	5000	1000	OK
5	4000	1000	OK
6	9000	500	OK
7	3000	2000	default

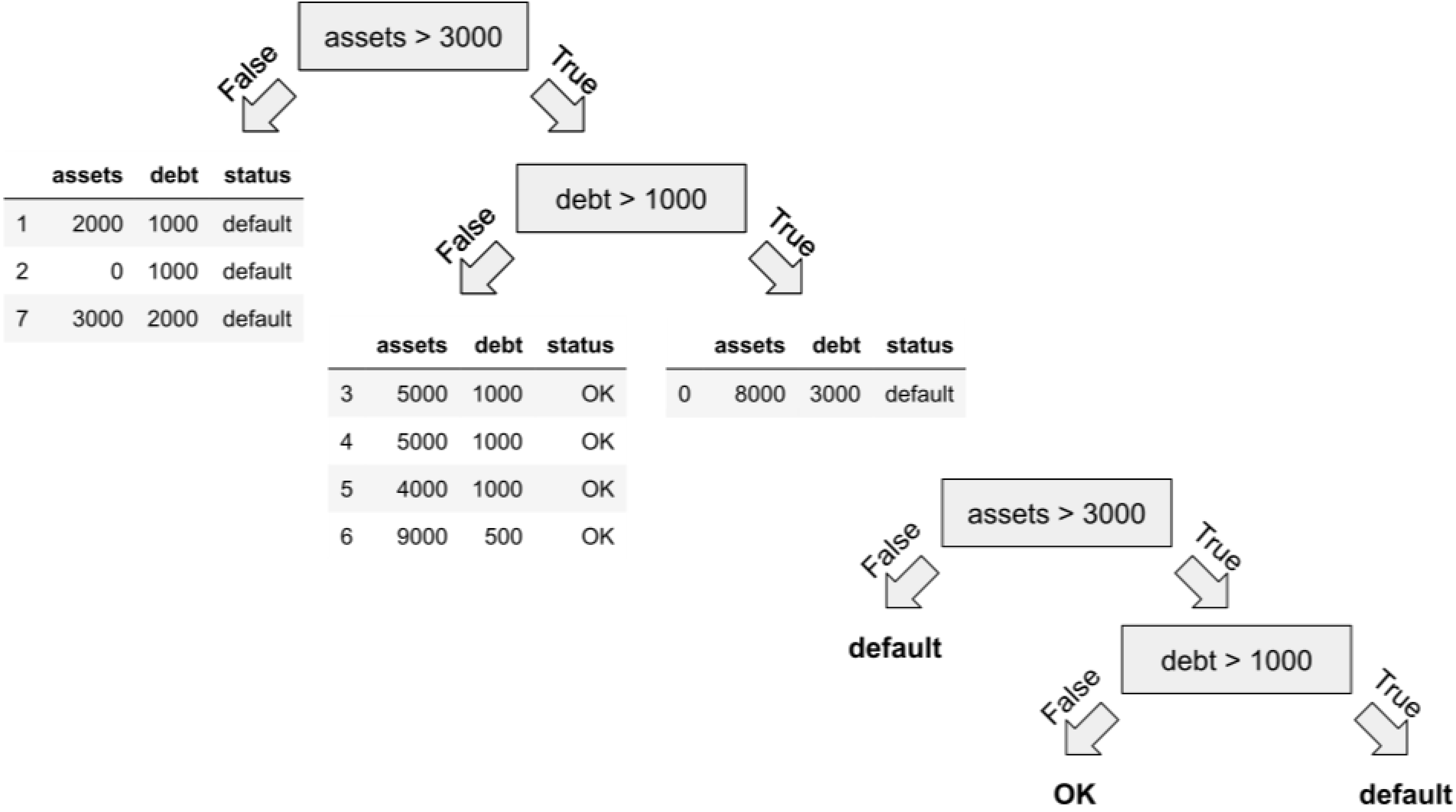
- For each feature, try all possible thresholds.
- For each threshold value T, measure the impurity of the split.
- Select the feature and the threshold with the lowest impurity possible.

Multi-Feature Tree

The best split is “assets > 3000”, which has the average impurity of 10%

	assets	debt	status
1	2000	1000	default
2	0	1000	default
7	3000	2000	default

	assets	debt	status
0	8000	3000	default
3	5000	1000	OK
4	5000	1000	OK
5	4000	1000	OK
6	9000	500	OK



assets > 3000
False
True
debt > 1000
False
True
assets debt status
1 2000 1000 default
2 0 1000 default
7 3000 2000 default
assets debt status
3 5000 1000 OK
4 5000 1000 OK
5 4000 1000 OK
6 9000 500 OK
assets debt status
0 8000 3000 default
assets > 3000
False
True
default
debt > 1000
False
True
OK
default

Stopping Criteria

- The group is already pure.
- The tree reached the depth limit (controlled by the max_depth parameter).
- The group is too small to continue splitting (controlled by the min_samples_leaf parameter).

Stopping Criteria

- Find the best split:
  - For each feature try all possible threshold values.
  - Use the one with the lowest impurity.
- If the maximal allowed depth is reached, stop.
- If the group on the left is sufficiently large and it’s not pure yet, repeat on the left.
- If the group on the right is sufficiently large and it’s not pure yet, repeat on the right.

Parameter Tuning

The two most Appropriate Parameters

- `max_depth`
- `min_leaf_size`

Iterate Using Multiple Depths

```
for depth in [1, 2, 3, 4, 5, 6, 10, 15, 20, None]:
    dt = DecisionTreeClassifier(max_depth=depth)
    dt.fit(X_train, y_train)
    y_pred = dt.predict_proba(X_val)[:, 1]
    auc = roc_auc_score(y_val, y_pred)
    print('%4s -> %.3f' % (depth, auc))
```

```
   1 -> 0.606
   2 -> 0.669
   3 -> 0.739
   4 -> 0.761
   5 -> 0.766
   6 -> 0.754
  10 -> 0.685
  15 -> 0.671
  20 -> 0.657
None -> 0.657
```

Optimal values for `max_depth`

Leaf Size

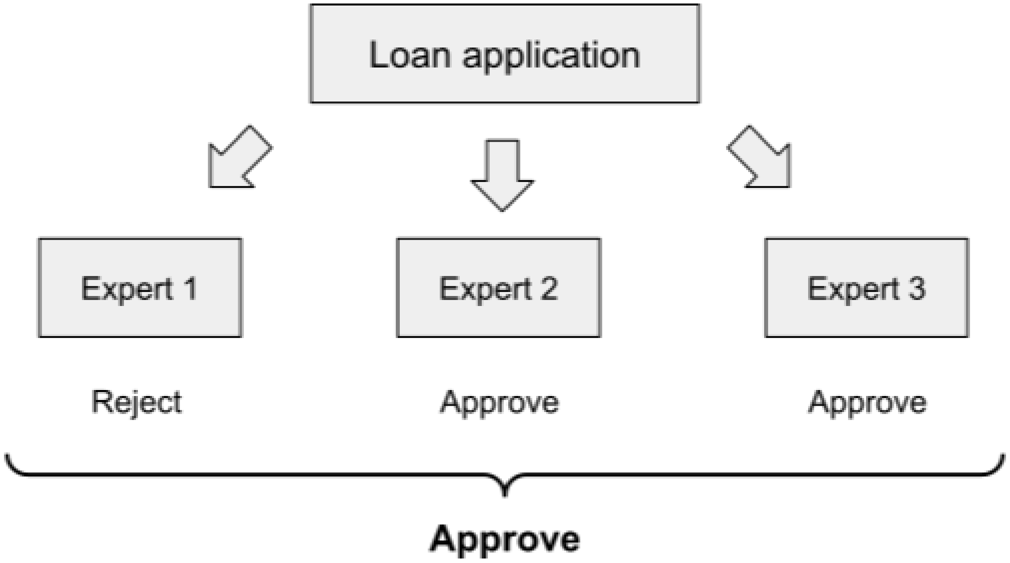
```
for m in [4, 5, 6]:
    print('depth: %s' % m)
    for s in [1, 5, 10, 15, 20, 50, 100, 200]:
        dt = DecisionTreeClassifier(max_depth=m, min_samples_leaf=s)
        dt.fit(X_train, y_train)
        y_pred = dt.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)
        print('%s -> %.3f' % (s, auc))
    print()
```

	depth=4	depth=5	depth=6
1	0.761	0.766	0.754
5	0.761	0.768	0.760
10	0.761	0.762	0.778
15	0.764	0.772	0.785
20	0.761	0.774	0.774
50	0.753	0.768	0.770
100	0.756	0.763	0.776
200	0.747	0.759	0.768

```
dt = DecisionTreeClassifier(max_depth=6,min_samples_leaf=15)
dt.fit(X_train, y_train)
```

Random Forest

How Would we Do this Manually?



Random Forest

Models we want to combine in an ensemble should not be the same. We can make sure they are different by training each tree on a different subset of features.

	assets	debt	price	status
0	8000	3000	9000	default
1	0	1000	1000	default
2	5000	1000	2500	OK
3	9000	500	3000	OK

	assets	debt	status
0	8000	3000	default
1	0	1000	default
2	5000	1000	OK
3	9000	500	OK

	debt	price	status
0	3000	9000	default
1	1000	1000	default
2	1000	2500	OK
3	500	3000	OK

	assets	price	status
0	8000	9000	default
1	0	1000	default
2	5000	2500	OK
3	9000	3000	OK

Tree 1

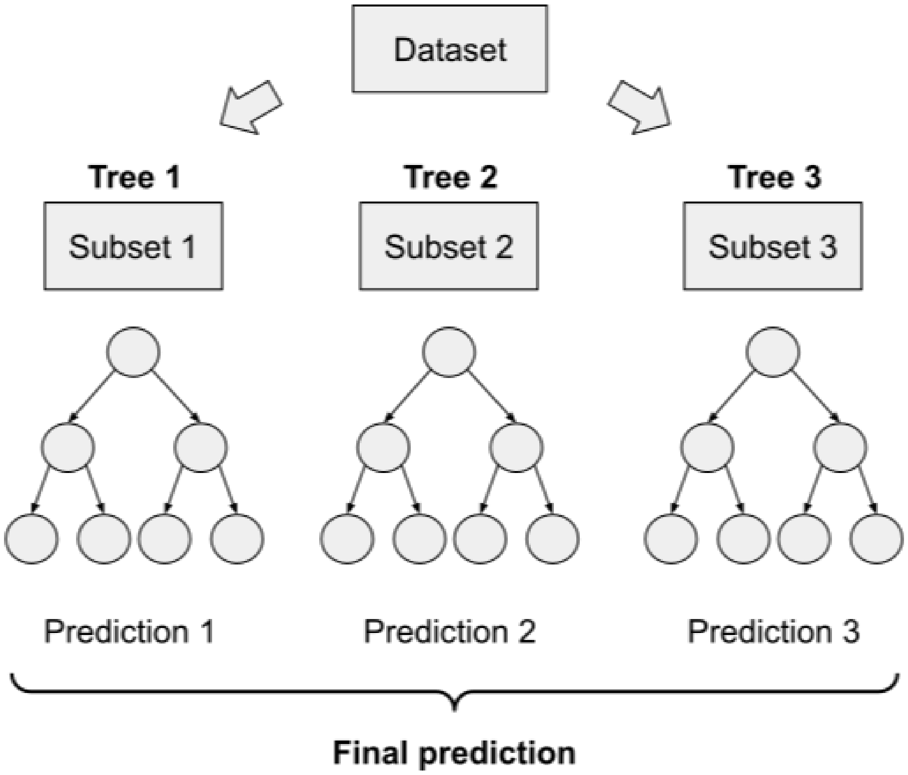
Tree 2

Tree 3

Ensemble

Training a Random Forest

- Train N independent decision tree models.
- For each model, select a random subset of features, and use only them for training.
- When predicting, combine the output of N models into one.



Training a Random Forest with Scikit-Learn

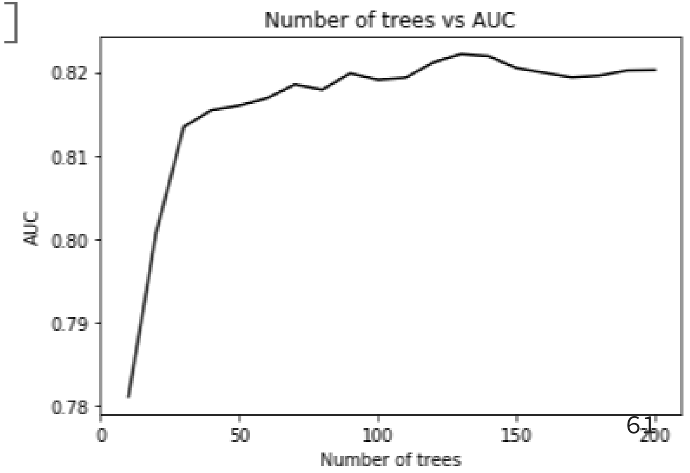
```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier(n_estimators=10)

rf.fit(X_train, y_train)

y_pred = rf.predict_proba(X_val)[:, 1]

roc_auc_score(y_val, y_pred)


# THE SCORE WILL RANGE FROM 77-80%... HENCE… RANDOM
```

Set the Random State

```
rf = RandomForestClassifier(n_estimators=10, random_state=3)

rf.fit(X_train, y_train)

y_pred = rf.predict_proba(X_val)[:, 1]

roc_auc_score(y_val, y_pred)
```

How Many Trees in the Forest?

```
aucs = []
for i in range(10, 201, 10):
    rf = RandomForestClassifier(n_estimators=i, random_state=3)
    rf.fit(X_train, y_train)
    y_pred = rf.predict_proba(X_val)[:, 1]
    auc = roc_auc_score(y_val, y_pred)
    print('%s -> %.3f' % (i, auc))
    aucs.append(auc)
plt.plot(range(10, 201, 10), aucs)
```



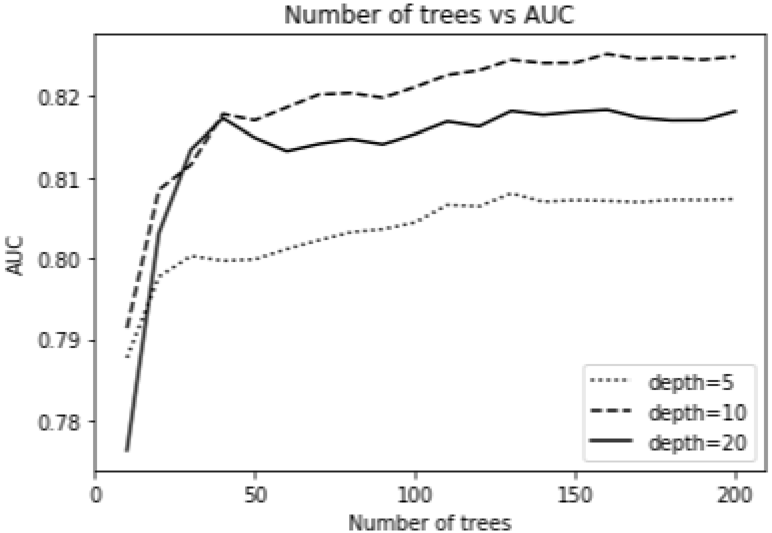
Parameter Tuning

Max Depth

```
all_aucs = {}
for depth in [5, 10, 20]:
    print('depth: %s' % depth)
    aucs = []
    for i in range(10, 201, 10):
        rf = RandomForestClassifier(n_estimators=i, max_depth=depth, random_state=1)
        rf.fit(X_train, y_train)
        y_pred = rf.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)
        print('%s -> %.3f' % (i, auc))
        aucs.append(auc)
    all_aucs[depth] = aucs
    print()
```

Random Forest Performance

```
num_trees = list(range(10, 201, 10))
plt.plot(num_trees, all_aucs[5], label='depth=5')
plt.plot(num_trees, all_aucs[10], label='depth=10')
plt.plot(num_trees, all_aucs[20], label='depth=20')
plt.legend()
```

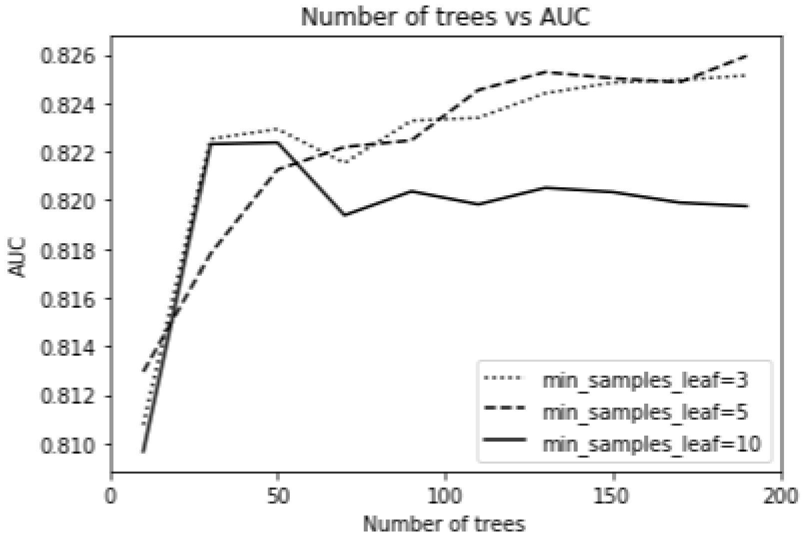


Min Sample Leafs

```
all_aucs = {}
 for m in [3, 5, 10]:
    print('min_samples_leaf: %s' % m)
    aucs = []
     for i in range(10, 201, 20):
        rf = RandomForestClassifier(n_estimators=i, max_depth=10, min_samples_leaf=m, random_state=1)
        rf.fit(X_train, y_train)
        y_pred = rf.predict_proba(X_val)[:, 1]
        auc = roc_auc_score(y_val, y_pred)
        print('%s -> %.3f' % (i, auc))
        aucs.append(auc)
     all_aucs[m] = aucs
    print()
```

Let's Plot it

```
num_trees = list(range(10, 201, 20))

plt.plot(num_trees, all_aucs[3], label='min_samples_leaf=3')

plt.plot(num_trees, all_aucs[5], label='min_samples_leaf=5')

plt.plot(num_trees, all_aucs[10],
label='min_samples_leaf=10')

plt.legend()
```



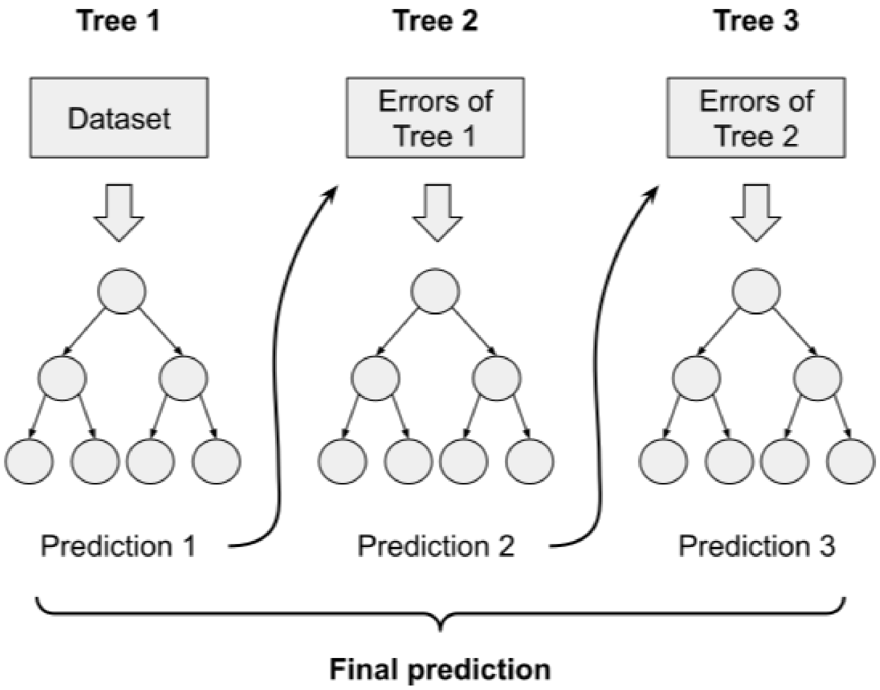
Let's Train the Final Model

```
rf = RandomForestClassifier(n_estimators=200, max_depth=10,
min_samples_leaf=5, random_state=1)
```

Gradient Boosting

Gradient Boosting

- Train the first model.
- Look at the errors it makes.
- Train another model that fixes these errors.
- Look at the errors again, repeat sequentially.



XGBoost (Extreme Gradient Boosting)

```
!pip install xgboost

import xgboost as xgb

dtrain = xgb.DMatrix(X_train, label=y_train,
feature_names=dv.feature_names_)

dval = xgb.DMatrix(X_val, label=y_val,
feature_names=dv.feature_names_)
```

Specify the Parameters for Training

```
xgb_params = {
    'eta': 0.3,
    'max_depth': 6,
    'min_child_weight': 1,

    'objective': 'binary:logistic',
    'nthread': 8,
    'seed': 1,
    'silent': 1
}
model = xgb.train(xgb_params, dtrain, num_boost_round=10)
```

Train an XGBoost Model

```
y_pred = model.predict(dval)
```

```
y_pred = model.predict(dval)
y_pred[:10]
```

```
array([0.08926772, 0.0468099 , 0.09692743, 0.17261842, 0.05435968,
       0.12576081, 0.08033007, 0.61870354, 0.486538  , 0.04056795],
      dtype=float32)
```

Compute the AUC

```
roc_auc_score(y_val, y_pred)
```

Summary

- Decision tree is a model that represents a sequence of if-then-else decisions.
- We train decision trees by selecting the best split using impurity measures.
- Random forest is a way to combine many decision trees in one model.
- A random forest should have a diverse set of models to make good predictions.
- The main parameters we need to change for random forest are the same as for decision trees: the depth and the maximal number of samples in each leaf.
- While in random forest the trees are independent, in gradient boosting the trees are sequential and each next model corrects the mistakes of the previous one.
- The parameters we need to tune for gradient boosting are similar to random forest: the depth, the maximal number of observations in the leaf and the number of trees.