11_ensembles

June 23, 2022

CPSC 330
Applied Machine Learning

1 Lecture 11: Ensembles

UBC 2022 Summer

Instructor: Mehrdad Oveisi

The interests of truth require a diversity of opinions.

by John Stuart Mill

1.1 Imports

[1]:
```
import os

%matplotlib inline
import string
import sys
from collections import deque

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd

sys.path.append("code/.")

from plotting_functions import *
from sklearn import datasets
from sklearn.compose import ColumnTransformer, make_column_transformer
from sklearn.dummy import DummyClassifier, DummyRegressor
from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor
from sklearn.impute import SimpleImputer
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.model_selection import (
    GridSearchCV,
    RandomizedSearchCV,
    cross_val_score,
    cross_validate,
    train_test_split,
)
from sklearn.pipeline import Pipeline, make_pipeline
from sklearn.preprocessing import OneHotEncoder, OrdinalEncoder, StandardScaler
from sklearn.svm import SVC, SVR
from sklearn.tree import DecisionTreeClassifier
from utils import *
```

1.2 Lecture learning objectives

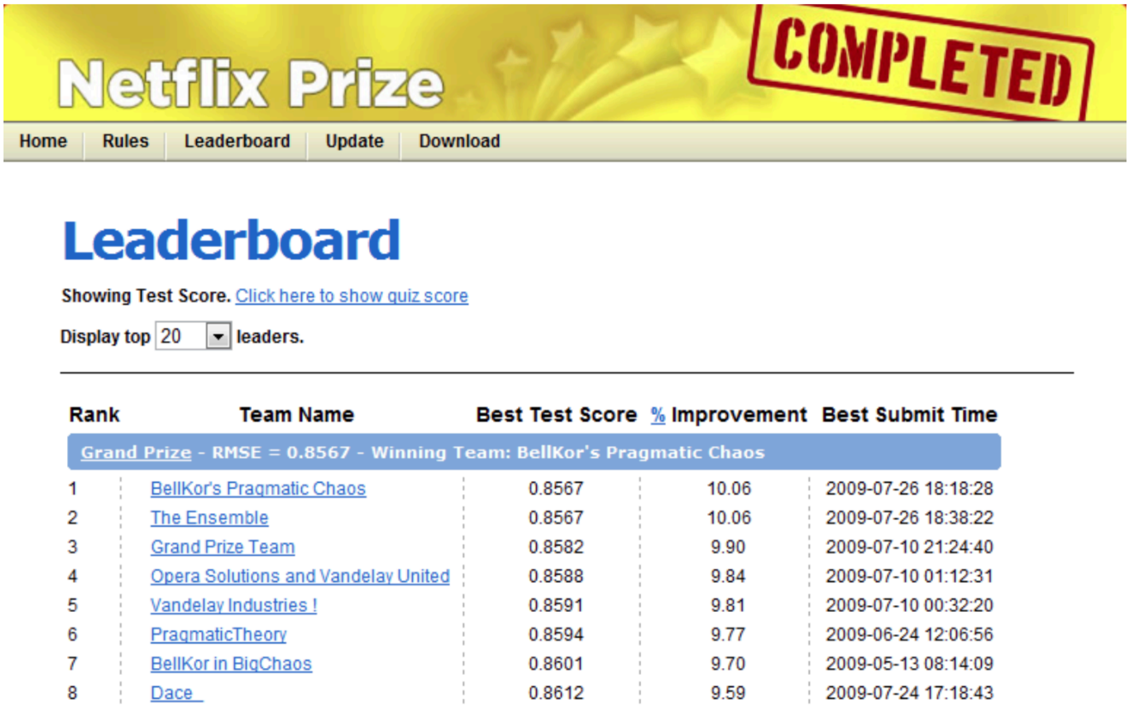
From this lecture, you will be able to

- Use `scikit-learn`'s `RandomForestClassifier` and explain its main hyperparameters.
- Explain randomness in random forest algorithm.
- Use other tree-based models such as `XGBoost` and `LGBM`.
- Employ ensemble classifier approaches, in particular model averaging and stacking.
- Explain voting and stacking and the differences between them.
- Use `scikit-learn` implementations of these ensemble methods.

1.3 Motivation

- **Ensembles** are models that combine multiple machine learning models to create more powerful models.

1.3.1 The Netflix prize



Rank	Team Name	Best Test Score	% Improvement	Best Submit Time
1	BellKor's Pragmatic Chaos	0.8567	10.06	2009-07-26 18:18:28
2	The Ensemble	0.8567	10.06	2009-07-26 18:38:22
3	Grand Prize Team	0.8582	9.90	2009-07-10 21:24:40
4	Opera Solutions and Vandelay United	0.8588	9.84	2009-07-10 01:12:31
5	Vandelay Industries !	0.8591	9.81	2009-07-10 00:32:20
6	PragmaticTheory	0.8594	9.77	2009-06-24 12:06:56
7	BellKor in BigChaos	0.8601	9.70	2009-05-13 08:14:09
8	Dace_	0.8612	9.59	2009-07-24 17:18:43

Source

- Most of the winning solutions for Kaggle competitions involve some kind of ensembling. For example:

IEEE-CIS Fraud Detection

Can you detect fraud from customer transactions?

Tags : tabular data, binary classification

S.No	Discussion Title
1	1st Place Solution - Part 1

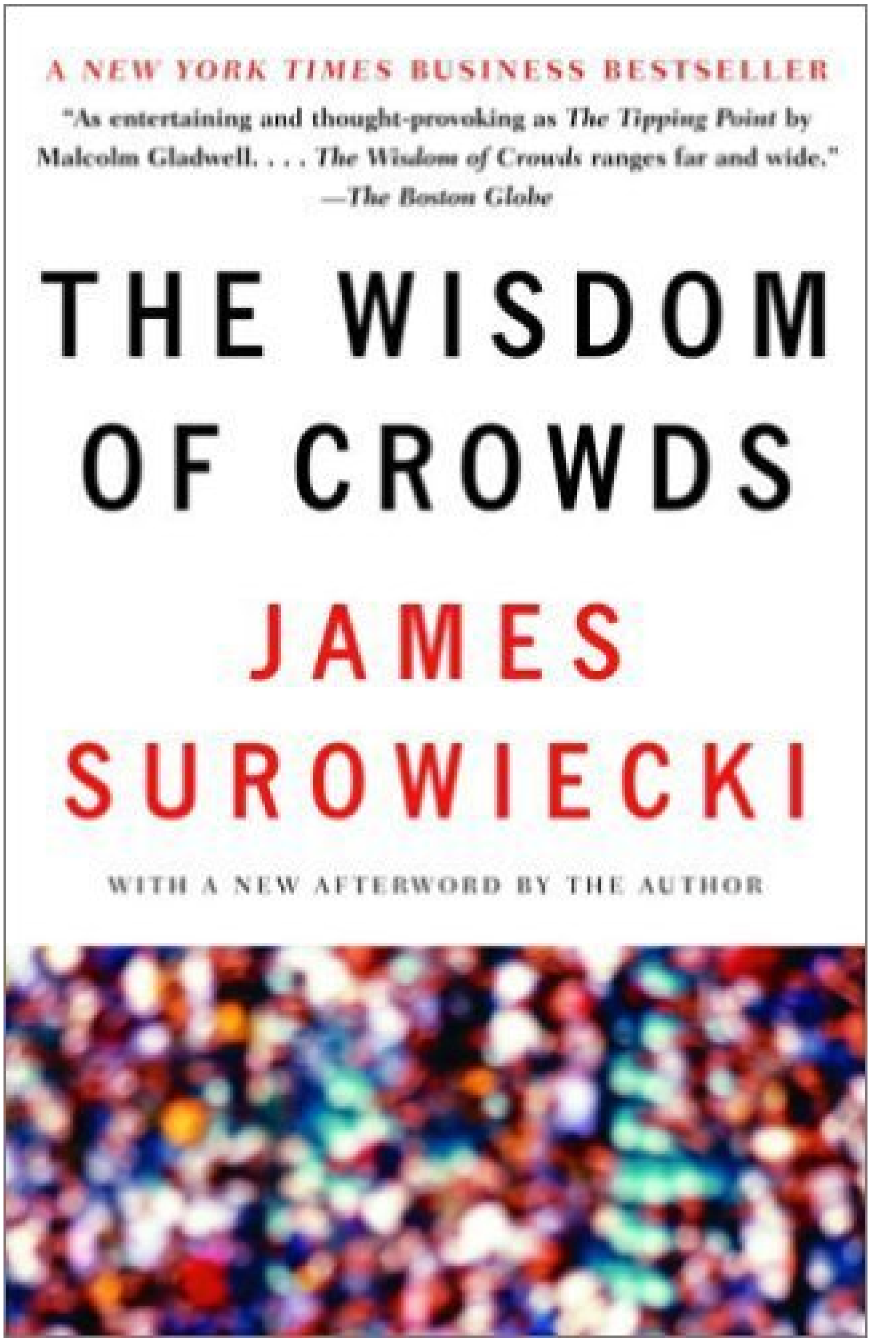
Models:

We had 3 main models (with single scores):

- Catboost (0.963915 public / 0.940826 private)
- LGBM (0.961748 / 0.938359)
- XGB (0.960205 / 0.932369)

Key idea: Groups can often make better decisions than individuals, especially when group members are diverse enough.

The Wisdom of Crowds



1.3.2 Tree-based ensemble models

- A number of ensemble models in ML literature.
- Most successful ones on a variety of datasets are tree-based models.
- We'll briefly talk about two such models:
  - Random forests
  - Gradient boosted trees
- We'll also talk about averaging and stacking.

1.3.3 Tree-based models

- Decision trees models are
  - **Interpretable**
  - They can capture **non-linear** relationships
  - They **do not** require **scaling** of the data and theoretically can work with **categorical** features
- But with a single decision trees are likely to **overfit**.

- Key idea: **Combine multiple trees** to build stronger models.
- These kinds of models are extremely popular in industry and machine learning competitions

1.3.4 Data

- Let's work with the adult census dataset. We had used this dataset in hw3 as well. So, if you still have the file, you can copy it (or add a symlink to it):

  cp hw/hw3/adult.csv lectures/data/

[2]:
```
adult_df_large = pd.read_csv("data/adult.csv")
train_df, test_df = train_test_split(adult_df_large, test_size=0.2,␣
↪random_state=42)
train_df_nan = train_df.replace("?", np.NaN)
test_df_nan = test_df.replace("?", np.NaN)
train_df_nan.head()
```

[2]:
```
       age  workclass  fnlwgt     education  education.num  \
5514    26    Private  256263       HS-grad              9
19777   24    Private  170277       HS-grad              9
10781   36    Private   75826     Bachelors             13
32240   22  State-gov   24395  Some-college             10
9876    31  Local-gov  356689     Bachelors             13

           marital.status      occupation   relationship   race     sex  \
5514        Never-married    Craft-repair  Not-in-family  White    Male
19777       Never-married   Other-service  Not-in-family  White  Female
10781            Divorced    Adm-clerical      Unmarried  White  Female
32240  Married-civ-spouse    Adm-clerical           Wife  White  Female
9876   Married-civ-spouse  Prof-specialty        Husband  White    Male

       capital.gain  capital.loss  hours.per.week native.country income
5514              0             0              25  United-States  <=50K
19777             0             0              35  United-States  <=50K
10781             0             0              40  United-States  <=50K
32240             0             0              20  United-States  <=50K
9876              0             0              40  United-States  <=50K
```

[3]:
```
numeric_features = ["age", "fnlwgt", "capital.gain", "capital.loss", "hours.per.
↪week"]
```

```
categorical_features = [
    "workclass",
    "marital.status",
    "occupation",
    "relationship",
    "native.country",
]
ordinal_features = ["education"]
binary_features = ["sex"]
drop_features = ["race", "education.num"]
target_column = "income"
```

[4]:
```
education_levels = [
    "Preschool",
    "1st-4th",
    "5th-6th",
    "7th-8th",
    "9th",
    "10th",
    "11th",
    "12th",
    "HS-grad",
    "Prof-school",
    "Assoc-voc",
    "Assoc-acdm",
    "Some-college",
    "Bachelors",
    "Masters",
    "Doctorate",
]
```

[5]:
```
assert set(education_levels) == set(train_df["education"].unique())
```

[6]:
```
numeric_transformer = make_pipeline(StandardScaler())

ordinal_transformer = make_pipeline(
    OrdinalEncoder(categories=[education_levels], dtype=int)
)

categorical_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OneHotEncoder(handle_unknown="ignore", sparse=False),
)

binary_transformer = make_pipeline(
    SimpleImputer(strategy="constant", fill_value="missing"),
    OneHotEncoder(drop="if_binary", dtype=int),
```

```
)

preprocessor = make_column_transformer(
    (numeric_transformer, numeric_features),
    (ordinal_transformer, ordinal_features),
    (binary_transformer, binary_features),
    (categorical_transformer, categorical_features),
    ("drop", drop_features),
)
```

[7]:
```
X_train = train_df_nan.drop(columns=[target_column])
y_train = train_df_nan[target_column]

X_test = test_df_nan.drop(columns=[target_column])
y_test = test_df_nan[target_column]
```

1.3.5 Do we have class imbalance?

- There is class imbalance. But without any context, both classes seem equally important.
- Let's use accuracy as our metric.

[8]:
```
train_df_nan["income"].value_counts(normalize=True)
```

[8]:
```
<=50K    0.757985
>50K     0.242015
Name: income, dtype: float64
```

[9]:
```
scoring_metric = "accuracy"
```

Let's store all the results in a dictionary called `results`.

[10]:
```
results = {}
```

1.3.6 Baselines

`DummyClassifier` baseline

[11]:
```
dummy = DummyClassifier(strategy="most_frequent")
results["Dummy"] = mean_std_cross_val_scores(
    dummy, X_train, y_train, return_train_score=True, scoring=scoring_metric
)
```

`DecisionTreeClassifier` baseline

- Let's try decision tree classifier on our data.

[12]:
```
pipe_dt = make_pipeline(preprocessor, DecisionTreeClassifier(random_state=123))
results["Decision tree"] = mean_std_cross_val_scores(
    pipe_dt, X_train, y_train, return_train_score=True, scoring=scoring_metric
)
```

```
pd.DataFrame(results).T
```

[12]:
```
                          fit_time         score_time         test_score  \
Dummy          0.013 (+/- 0.004)  0.010 (+/- 0.003)  0.758 (+/- 0.000)
Decision tree  0.281 (+/- 0.022)  0.026 (+/- 0.001)  0.813 (+/- 0.003)

                     train_score
Dummy          0.758 (+/- 0.000)
Decision tree  1.000 (+/- 0.000)
```

Decision tree is clearly **overfitting**.

1.4 Random forests

1.4.1 General idea

- A single decision tree is likely to overfit
- Use a **collection of diverse** decision trees
- Each tree overfits on some part of the data but we can reduce overfitting by averaging the results
  - can be shown mathematically

1.4.2 `RandomForestClassifier`

- Before understanding the details let's first try it out.

[13]:
```
from sklearn.ensemble import RandomForestClassifier

pipe_rf = make_pipeline(
    preprocessor, RandomForestClassifier(random_state=123, n_jobs=-1)
)
results["Random forests"] = mean_std_cross_val_scores(
    pipe_rf, X_train, y_train, return_train_score=True, scoring=scoring_metric
)
pd.DataFrame(results).T
```

[13]:
```
                           fit_time         score_time         test_score  \
Dummy           0.013 (+/- 0.004)  0.010 (+/- 0.003)  0.758 (+/- 0.000)
Decision tree   0.281 (+/- 0.022)  0.026 (+/- 0.001)  0.813 (+/- 0.003)
Random forests  1.822 (+/- 0.977)  0.171 (+/- 0.030)  0.857 (+/- 0.004)

                      train_score
Dummy           0.758 (+/- 0.000)
Decision tree   1.000 (+/- 0.000)
Random forests  1.000 (+/- 0.000)
```

The validation scores are better although it seems likes we are still overfitting.

1.4.3 How do they work?

- Decide how many decision trees we want to build
  - can control with `n_estimators` hyperparameter
- `fit` a **diverse set** of that many decision trees by **injecting randomness** in the classifier construction
- `predict` by **voting** (classification) or **averaging** (regression) of predictions given by individual models

1.4.4 Inject randomness in the classifier construction

To ensure that the trees in the random forest are different we inject randomness in two ways:

1. Data: **Build each tree on a bootstrap sample** (i.e., a sample drawn **with replacement** from the training set)
2. Features: **At each node, select a random subset of features** (controlled by `max_features` in `scikit-learn`) and look for the best possible test involving one of these features

An example of a bootstrap samples Suppose this is your original dataset: [1,2,3,4] - a sample drawn with replacement: [1,1,3,4] - a sample drawn with replacement: [3,2,2,2] - a sample drawn with replacement: [1,2,4,4] - …

See Also (Optional) There is also something called `ExtraTreesClassifier`, where we add more randomness by consider a random subset of features at each split and **random threshold**.

1.4.5 The random forests classifier

Training time: - Create a collection (**ensemble**) of trees. - Grow each tree on an independent **bootstrap sample from the data**. - At each node: - Randomly select a **subset of features** out of all features (independently for each node). - Find the **best split** on the selected features. - Grow the trees to **maximum depth**.

Prediction time: - **Vote** the trees to get predictions for new example.

1.4.6 Example

- Let's create a random forest with 3 estimators.
- I'm using `max_depth=2` for easy visualization.

[14]:
```
pipe_rf_demo = make_pipeline(
    preprocessor, RandomForestClassifier(max_depth=2, n_estimators=3,
 ↪random_state=123)
)
pipe_rf_demo.fit(X_train, y_train);
```

- Let's get the feature names of transformed features.

[15]:
```
feature_names = (
    numeric_features
    + ordinal_features
    + binary_features
```

```
    + list(
        pipe_rf_demo.named_steps["columntransformer"]
        .named_transformers_["pipeline-4"]
        .named_steps["onehotencoder"]
        .get_feature_names_out()
    )
)
pd.DataFrame(columns=feature_names)  # Take a look at the feature names
```

[15]:
```
Empty DataFrame
Columns: [age, fnlwgt, capital.gain, capital.loss, hours.per.week, education,
sex, x0_Federal-gov, x0_Local-gov, x0_Never-worked, x0_Private, x0_Self-emp-inc,
x0_Self-emp-not-inc, x0_State-gov, x0_Without-pay, x0_missing, x1_Divorced,
x1_Married-AF-spouse, x1_Married-civ-spouse, x1_Married-spouse-absent, x1_Never-
married, x1_Separated, x1_Widowed, x2_Adm-clerical, x2_Armed-Forces, x2_Craft-
repair, x2_Exec-managerial, x2_Farming-fishing, x2_Handlers-cleaners,
x2_Machine-op-inspct, x2_Other-service, x2_Priv-house-serv, x2_Prof-specialty,
x2_Protective-serv, x2_Sales, x2_Tech-support, x2_Transport-moving, x2_missing,
x3_Husband, x3_Not-in-family, x3_Other-relative, x3_Own-child, x3_Unmarried,
x3_Wife, x4_Cambodia, x4_Canada, x4_China, x4_Columbia, x4_Cuba, x4_Dominican-
Republic, x4_Ecuador, x4_El-Salvador, x4_England, x4_France, x4_Germany,
x4_Greece, x4_Guatemala, x4_Haiti, x4_Holand-Netherlands, x4_Honduras, x4_Hong,
x4_Hungary, x4_India, x4_Iran, x4_Ireland, x4_Italy, x4_Jamaica, x4_Japan,
x4_Laos, x4_Mexico, x4_Nicaragua, x4_Outlying-US(Guam-USVI-etc), x4_Peru,
x4_Philippines, x4_Poland, x4_Portugal, x4_Puerto-Rico, x4_Scotland, x4_South,
x4_Taiwan, x4_Thailand, x4_Trinadad&Tobago, x4_United-States, x4_Vietnam,
x4_Yugoslavia, x4_missing]
Index: []

[0 rows x 86 columns]
```

- Let's sample a test example.

[16]:
```
test_example = X_test.sample(1)
print("Classes: ", pipe_rf_demo.classes_)
print("Prediction by random forest: ", pipe_rf_demo.predict(test_example))
transformed_example = preprocessor.transform(test_example)
pd.DataFrame(transformed_example, columns=feature_names)
```

```
Classes:  ['<=50K' '>50K']
Prediction by random forest:  ['<=50K']
```

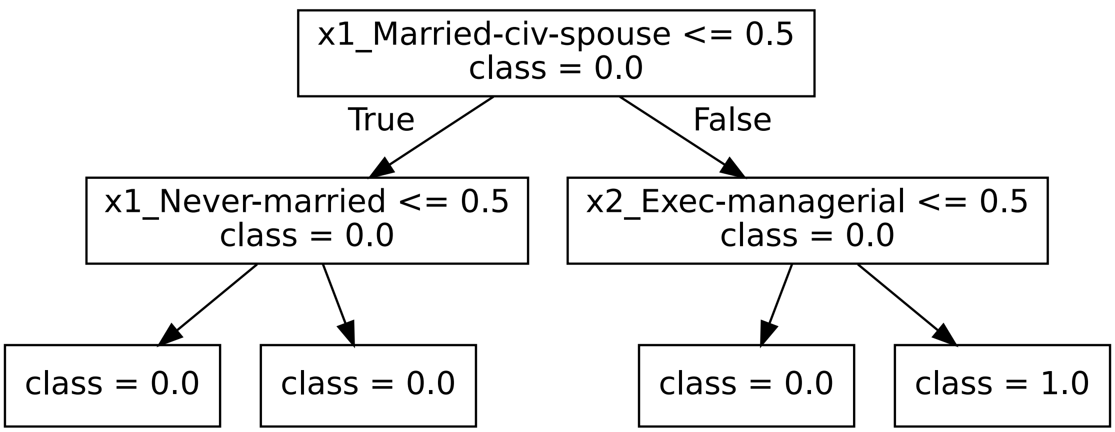
[16]:
```
        age    fnlwgt  capital.gain  capital.loss  hours.per.week  education  \
0  1.138787 -0.037063     -0.147166      -0.21768       -2.069258        4.0

   sex  x0_Federal-gov  x0_Local-gov  x0_Never-worked  …  x4_Puerto-Rico  \
0  0.0             0.0           0.0              0.0  …             0.0
```

```
   x4_Scotland  x4_South  x4_Taiwan  x4_Thailand  x4_Trinadad&Tobago  \
0          0.0       0.0        0.0          0.0                 0.0

   x4_United-States  x4_Vietnam  x4_Yugoslavia  x4_missing
0               1.0         0.0            0.0         0.0

[1 rows x 86 columns]
```

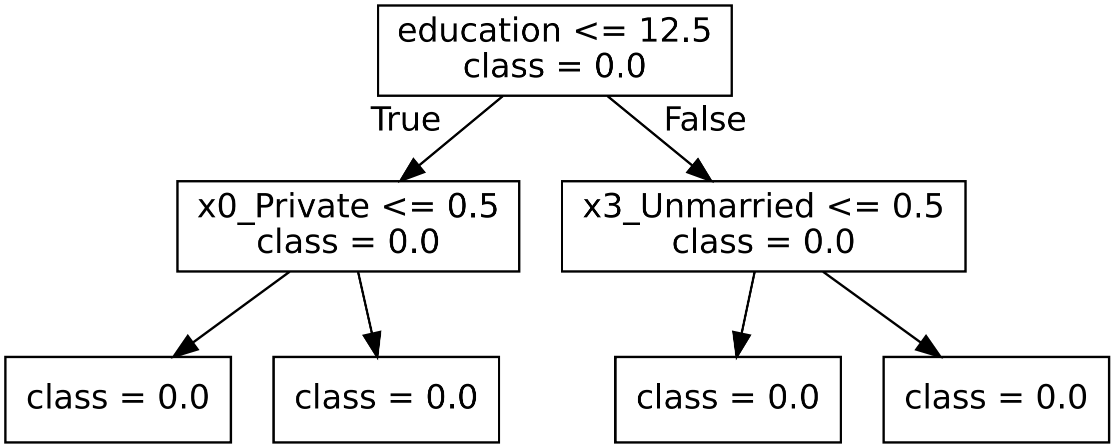
- We can look at **different trees** created by random forest.
- Note that each tree looks at **different set of features** and slightly **different data**.

[17]:
```python
for i, tree in enumerate(
    pipe_rf_demo.named_steps["randomforestclassifier"].estimators_
):
    print("\n\nTree", i + 1)
    display(display_tree(feature_names, tree))
    print("\nPrediction:", tree.predict(preprocessor.transform(test_example)))
```

```
Tree 1
```

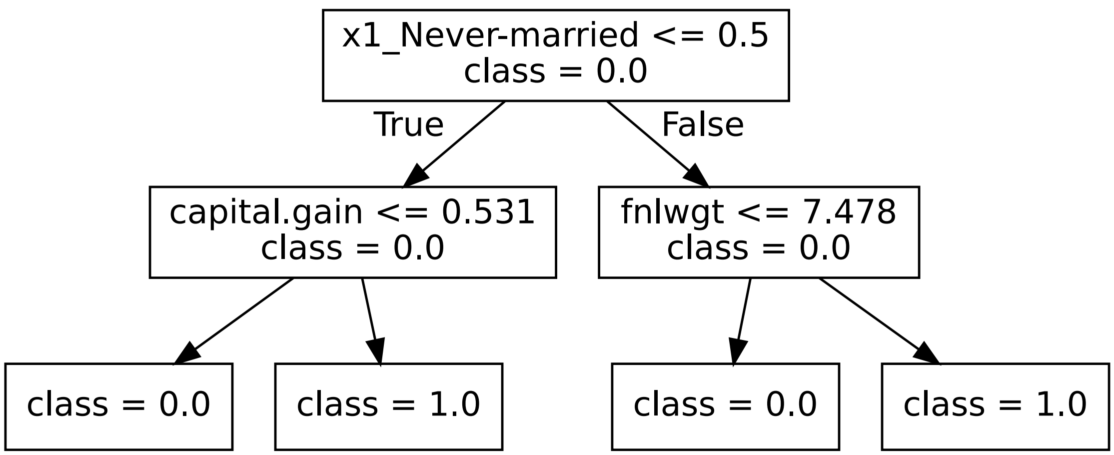



```
Prediction: [0.]


Tree 2
```




```
Prediction: [0.]
```

```
Tree 3
```



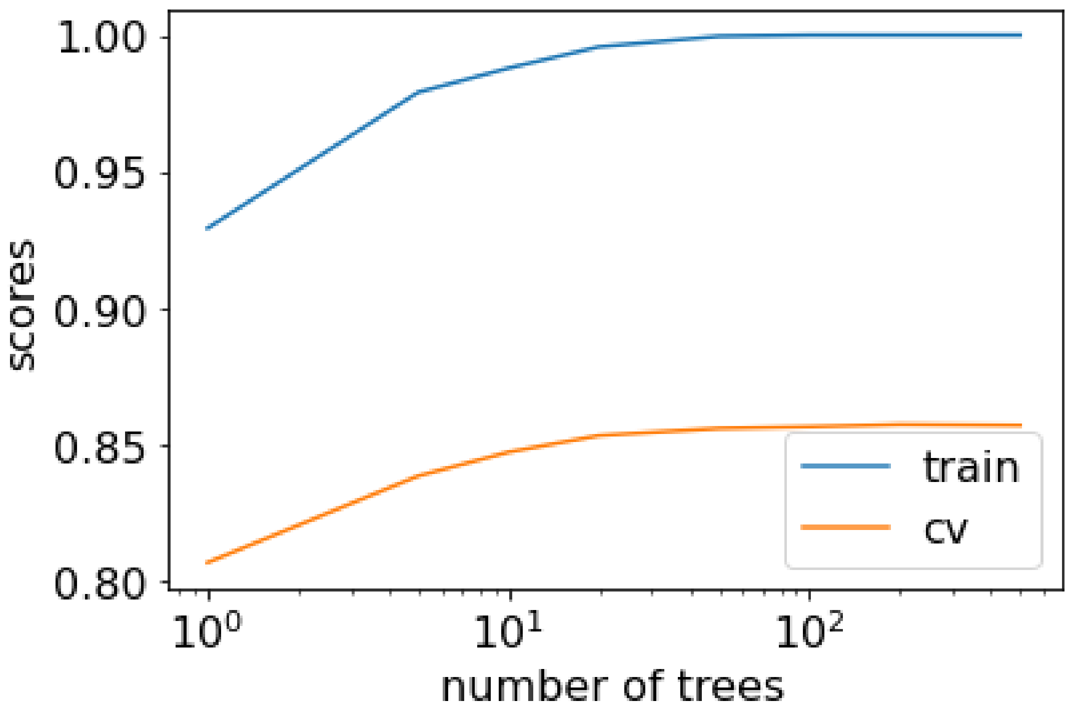

```
Prediction: [0.]
```

1.4.7 Some important hyperparameters:

- `n_estimators`: number of decision trees (higher = more complexity)
- `max_depth`: max depth of each decision tree (higher = more complexity)
- `max_features`: the number of features you get to look at each split (higher = more complexity)

1.4.8 Random forests: number of trees (n_estimators) and the fundamental tradeoff

[18]:
```
make_num_tree_plot(
    preprocessor, X_train, y_train, X_test, y_test, [1, 5, 10, 20, 50, 100,␣
 ↪200, 500]
)
```



Number of trees and fundamental trade-off

- Above: seems like we're beating the fundamental "tradeoff" by **increasing training score** and **not decreasing validation score** much.
- This is the promise of ensembles, though it's not guaranteed to work so nicely.

More trees are always better! We pick less trees for speed.

1.4.9 Strengths

- Usually **one of the best** performing **off-the-shelf** classifiers without heavy tuning of hyperparameters
- **Don't** require **scaling** of data
- **Less** likely to **overfit**
- Slower than decision trees because we are fitting multiple trees but **can easily parallelize training** because all trees are independent of each other
- In general, able to capture a much broader picture of the data compared to a single decision tree.

1.4.10 Weaknesses

- Require more memory
- Hard to interpret
- Tend not to perform well on high dimensional sparse data such as text data

Important Make sure to set the `random_state` for reproducibility. Changing the `random_state` can have a big impact on the model and the results due to the random nature of these models. Having more trees can get you a more robust estimate.

See Also (Optional) The original random forests paper by Leo Breiman.

1.5 Gradient boosted trees

Another popular and effective class of tree-based models is gradient boosted trees.

- **No** randomization
- The key idea is:combining many simple models called **weak learners to create a strong learner**
- They combine **multiple shallow** (depth 1 to 5) decision trees
- They build trees in a **serial manner**, where each tree tries to **correct the mistakes** of the previous one

1.5.1 Important hyperparameters

- `n_estimators`
  - control the number of trees to build
- `learning_rate`
  - controls **how strongly** each tree tries to **correct the mistakes** of the previous trees
  - **higher** `learning_rate`
    - means each tree can make **stronger corrections**,
    - which means **more complex** model

We'll not go into the details. We'll look at brief examples of using the following three gradient boosted tree models.

- XGBoost
- LightGBM
- CatBoost

1.5.2 XGBoost

- Not part of `sklearn` but has similar interface.
- Install it in your conda environment: `conda install -n cpsc330 -c conda-forge xgboost`
- Supports missing values
- GPU training, networked parallel training
- Supports sparse data
- Typically better scores than random forests

1.5.3 LightGBM

- Not part of `sklearn` but has similar interface.
- Install it in your conda environment: `conda install -n cpsc330 -c conda-forge lightgbm`
- Small model size
- Faster
- Typically better scores than random forests

1.5.4 CatBoost

- Not part of `sklearn` but has similar interface.
- Install it in your conda environment: `conda install -n cpsc330 -c conda-forge catboost`
- Usually better scores but slower compared to `XGBoost` and `LightGBM`

[19]:
```
import warnings

warnings.simplefilter(action="ignore", category=FutureWarning)
warnings.simplefilter(action="ignore", category=UserWarning)
```

[20]:
```
from catboost import CatBoostClassifier
from lightgbm.sklearn import LGBMClassifier
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier

pipe_lr = make_pipeline(
    preprocessor, LogisticRegression(max_iter=2000, random_state=123)
)
pipe_dt = make_pipeline(preprocessor, DecisionTreeClassifier(random_state=123))
pipe_rf = make_pipeline(preprocessor, RandomForestClassifier(random_state=123))
pipe_xgb = make_pipeline(
    preprocessor, XGBClassifier(random_state=123, eval_metric="logloss",␣
↪verbosity=0)
)
pipe_lgbm = make_pipeline(preprocessor, LGBMClassifier(random_state=123))
pipe_catboost = make_pipeline(
    preprocessor, CatBoostClassifier(verbose=0, random_state=123)
)
classifiers = {
    "logistic regression": pipe_lr,
    "decision tree": pipe_dt,
    "random forest": pipe_rf,
    "XGBoost": pipe_xgb,
    "LightGBM": pipe_lgbm,
    "CatBoost": pipe_catboost,
}
```

[21]:
```python
results = {}
```

[22]:
```python
dummy = DummyClassifier(strategy="most_frequent")
results["Dummy"] = mean_std_cross_val_scores(
    dummy, X_train, y_train, return_train_score=True, scoring=scoring_metric
)
```

[23]:
```python
for (name, model) in classifiers.items():
    results[name] = mean_std_cross_val_scores(
        model, X_train, y_train, return_train_score=True, scoring=scoring_metric
    )
```

[24]:
```python
pd.DataFrame(results).T
```

[24]:
```
                                fit_time          score_time          test_score  \
Dummy                    0.016 (+/- 0.006)  0.012 (+/- 0.004)  0.758 (+/- 0.000)
logistic regression      2.360 (+/- 0.338)  0.044 (+/- 0.016)  0.850 (+/- 0.006)
decision tree            0.343 (+/- 0.031)  0.031 (+/- 0.002)  0.813 (+/- 0.003)
random forest            2.409 (+/- 0.367)  0.151 (+/- 0.011)  0.857 (+/- 0.004)
XGBoost                  6.625 (+/- 6.009)  0.079 (+/- 0.007)  0.870 (+/- 0.003)
LightGBM                 0.376 (+/- 0.047)  0.074 (+/- 0.006)  0.871 (+/- 0.004)
CatBoost                 8.199 (+/- 0.663)  0.215 (+/- 0.035)  0.872 (+/- 0.003)

                             train_score
Dummy                    0.758 (+/- 0.000)
logistic regression      0.851 (+/- 0.001)
decision tree            1.000 (+/- 0.000)
random forest            1.000 (+/- 0.000)
XGBoost                  0.909 (+/- 0.002)
LightGBM                 0.892 (+/- 0.000)
CatBoost                 0.900 (+/- 0.001)
```

Some observations - Keep in mind all these results are with default hyperparameters - Ideally we would carry out hyperparameter optimization for all of them and then compare the results. - We are using a particular scoring metric (accuracy in this case) - We are scaling numeric features but it shouldn't matter for these tree-based models. - Look at the std. Doesn't look very high. - The scores look more or less stable.

[25]:
```python
pd.DataFrame(results).T.sort_values('test_score')
```

[25]:
```
                                fit_time          score_time          test_score  \
Dummy                    0.016 (+/- 0.006)  0.012 (+/- 0.004)  0.758 (+/- 0.000)
decision tree            0.343 (+/- 0.031)  0.031 (+/- 0.002)  0.813 (+/- 0.003)
logistic regression      2.360 (+/- 0.338)  0.044 (+/- 0.016)  0.850 (+/- 0.006)
random forest            2.409 (+/- 0.367)  0.151 (+/- 0.011)  0.857 (+/- 0.004)
XGBoost                  6.625 (+/- 6.009)  0.079 (+/- 0.007)  0.870 (+/- 0.003)
LightGBM                 0.376 (+/- 0.047)  0.074 (+/- 0.006)  0.871 (+/- 0.004)
CatBoost                 8.199 (+/- 0.663)  0.215 (+/- 0.035)  0.872 (+/- 0.003)
```

```
                        train_score
Dummy                  0.758 (+/- 0.000)
decision tree          1.000 (+/- 0.000)
logistic regression    0.851 (+/- 0.001)
random forest          1.000 (+/- 0.000)
XGBoost                0.909 (+/- 0.002)
LightGBM               0.892 (+/- 0.000)
CatBoost               0.900 (+/- 0.001)
```

[26]:

```
# comparison of results (excluding the the 'Dummy' row [1:])

cv_score_order = pd.DataFrame(results).T[1:].sort_values('test_score').index
print('\nCV scores:')
print(*cv_score_order, sep=' < ')

fit_time_order = pd.DataFrame(results).T[1:].sort_values('fit_time').index
print('\nFitting speeds:')
print(*fit_time_order, sep=' > ')
```

```
CV scores:
decision tree < logistic regression < random forest < XGBoost < LightGBM <
CatBoost

Fitting speeds:
decision tree > LightGBM > logistic regression > random forest > XGBoost >
CatBoost
```

- Decision trees and random forests overfit
  - Other models do not seem to overfit much.
- Fit times
  - Decision trees are fast but not very accurate
  - LightGBM is faster than decision trees and more accurate!
  - CatBoost fit time is highest followed by random forests.

  - There is not much difference between the validation scores of XGBoost, LightGBM, and CatBoost but it is about 48x slower than LightGBM!
  - XGBoost and LightGBM are faster and more accurate than random forest!

- Scores times
  - Prediction times are much smaller in all cases.

1.5.5 What classifier should I use?

Simple answer - Whichever gets the highest CV score making sure that you're not overusing the validation set.

Interpretability - This is an area of growing interest and concern in ML. - How important is interpretability for you? - In the next class we'll talk about interpretability of non-linear models.

Speed/code maintenance - Other considerations could be speed (fit and/or predict), maintainability of the code.

Finally, you could use all of them!

1.6 Averaging

Earlier we looked at a bunch of classifiers:

[27]:
```
print(*classifiers, sep=', ')
```

```
logistic regression, decision tree, random forest, XGBoost, LightGBM, CatBoost
```

What if we use all these models and let them vote during prediction time?

[28]:
```
from sklearn.ensemble import VotingClassifier

averaging_model = VotingClassifier(
    list(classifiers.items()), voting="soft"
)  # need the list() here for cross_val to work!
```

[29]:
```
from sklearn import set_config

set_config(display="diagram")  # global setting
```

[30]:
```
averaging_model
```

[30]:
```
VotingClassifier(estimators=[('logistic regression',
                              Pipeline(steps=[('columntransformer',
ColumnTransformer(transformers=[('pipeline-1',
Pipeline(steps=[('standardscaler',
                 StandardScaler())]),
['age',
 'fnlwgt',
 'capital.gain',
 'capital.loss',
 'hours.per.week']),
('pipeline-2',
Pipeline(steps=[('ordinalencoder',
                 OrdinalEncoder(categories=[['Preschool',
                                             '1st-4th'…
Pipeline(steps=[('simpleimputer',
                 SimpleImputer(fill_value='missing',
                               strategy='constant')),
                ('onehotencoder',
                 OneHotEncoder(handle_unknown='ignore',
                               sparse=False))]),
['workclass',
```

```
 'marital.status',
 'occupation',
 'relationship',
 'native.country']),
('drop',
'drop',
['race',
 'education.num'])])),
                                              ('catboostclassifier',
                                               <catboost.core.CatBoostClassifier
object at 0x7fd6e5898310>)]))],
                 voting='soft')
```

This `VotingClassifier` will take a *vote* using the predictions of the constituent classifier pipelines.

Main parameter: `voting` - `voting='hard'` - it uses the output of `predict` and actually votes. - `voting='soft'` - with `voting='soft'` it **averages the output** of `predict_proba` and then thresholds / takes the larger.

- The choice depends on whether you trust `predict_proba` from your base classifiers - if so, it's nice to access that information.

[31]:
```
averaging_model.fit(X_train, y_train);
```

- What happens when you `fit` a `VotingClassifier`?
  - It will fit all constituent models.

Note It seems sklearn requires us to actually call `fit` on the `VotingClassifier`, instead of passing in pre-fit models. This is an implementation choice rather than a conceptual limitation.

Let's look at particular test examples where `income` is ">50k" (y=1):

[32]:
```
test_g50k = (test_df.query("income == '>50K'")
             .sample(4, random_state=2)
             .drop(columns=["income"]))

test_l50k = (test_df.query("income == '<=50K'")
             .sample(4, random_state=2)
             .drop(columns=["income"]))
```

[33]:
```
averaging_model.classes_
```

[33]:
```
array(['<=50K', '>50K'], dtype=object)
```

[34]:
```
voting = {"Voting classifier": averaging_model.predict(test_g50k)}
pd.DataFrame(voting)
```

[34]:
```
  Voting classifier
0              >50K
1              >50K
```

```
2               >50K
3              <=50K
```

For hard voting, these are the votes:

[35]:
```
hard = {
    name: classifier.predict(test_g50k)
    for name, classifier in averaging_model.named_estimators_.items()
}
hard.update(voting)
pd.DataFrame(hard)
```

[35]:
```
   logistic regression  decision tree  random forest  XGBoost  LightGBM  \
0                    1              1              1        1         1
1                    1              1              1        1         1
2                    1              0              1        1         1
3                    0              0              0        0         0

   CatBoost Voting classifier
0         1              >50K
1         1              >50K
2         1              >50K
3         0             <=50K
```

For soft voting, these are the scores:

[36]:
```
soft = {
    name: classifier.predict_proba(test_g50k)[:,1]
    for name, classifier in averaging_model.named_estimators_.items()
}
soft.update(voting)
pd.DataFrame(soft)
```

[36]:
```
   logistic regression  decision tree  random forest   XGBoost  LightGBM  \
0             1.000000            1.0           1.00  0.998319  0.998124
1             0.581298            1.0           0.69  0.699329  0.712771
2             0.503490            0.0           0.63  0.681060  0.715427
3             0.112505            0.0           0.43  0.210991  0.190440

   CatBoost Voting classifier
0  0.998798              >50K
1  0.732028              >50K
2  0.679881              >50K
3  0.232039             <=50K
```

(Aside: the probability scores from `DecisionTreeClassifier` are pretty bad)

Let's see how well this model performs.

[37]:
```
results["Voting"] = mean_std_cross_val_scores(averaging_model, X_train, y_train)
```

[38]:
```
pd.DataFrame(results).T
```

[38]:
```
                               fit_time            score_time            test_score  \
Dummy                     0.016 (+/- 0.006)   0.012 (+/- 0.004)   0.758 (+/- 0.000)
logistic regression       2.360 (+/- 0.338)   0.044 (+/- 0.016)   0.850 (+/- 0.006)
decision tree             0.343 (+/- 0.031)   0.031 (+/- 0.002)   0.813 (+/- 0.003)
random forest             2.409 (+/- 0.367)   0.151 (+/- 0.011)   0.857 (+/- 0.004)
XGBoost                   6.625 (+/- 6.009)   0.079 (+/- 0.007)   0.870 (+/- 0.003)
LightGBM                  0.376 (+/- 0.047)   0.074 (+/- 0.006)   0.871 (+/- 0.004)
CatBoost                  8.199 (+/- 0.663)   0.215 (+/- 0.035)   0.872 (+/- 0.003)
Voting                   20.037 (+/- 3.541)   0.554 (+/- 0.019)   0.868 (+/- 0.003)

                           train_score
Dummy                0.758 (+/- 0.000)
logistic regression  0.851 (+/- 0.001)
decision tree        1.000 (+/- 0.000)
random forest        1.000 (+/- 0.000)
XGBoost              0.909 (+/- 0.002)
LightGBM             0.892 (+/- 0.000)
CatBoost             0.900 (+/- 0.001)
Voting                             NaN
```

It appears that here we didn't do much better than our best classifier :(

Let's try removing decision tree classifier.

[39]:
```
classifiers_ndt = classifiers.copy()
del classifiers_ndt["decision tree"]
averaging_model_ndt = VotingClassifier(
    list(classifiers_ndt.items()), voting="soft"
)  # need the list() here for cross_val to work!

results["Voting_ndt"] = mean_std_cross_val_scores(
    averaging_model_ndt,
    X_train,
    y_train,
    return_train_score=True,
    scoring=scoring_metric,
)
```

[40]:
```
pd.DataFrame(results).T
```

[40]:
```
                               fit_time            score_time            test_score  \
Dummy                     0.016 (+/- 0.006)   0.012 (+/- 0.004)   0.758 (+/- 0.000)
logistic regression       2.360 (+/- 0.338)   0.044 (+/- 0.016)   0.850 (+/- 0.006)
decision tree             0.343 (+/- 0.031)   0.031 (+/- 0.002)   0.813 (+/- 0.003)
```

```
random forest         2.409 (+/- 0.367)  0.151 (+/- 0.011)  0.857 (+/- 0.004)
XGBoost               6.625 (+/- 6.009)  0.079 (+/- 0.007)  0.870 (+/- 0.003)
LightGBM              0.376 (+/- 0.047)  0.074 (+/- 0.006)  0.871 (+/- 0.004)
CatBoost              8.199 (+/- 0.663)  0.215 (+/- 0.035)  0.872 (+/- 0.003)
Voting               20.037 (+/- 3.541)  0.554 (+/- 0.019)  0.868 (+/- 0.003)
Voting_ndt           18.040 (+/- 3.314)  0.611 (+/- 0.053)  0.872 (+/- 0.003)

                           train_score
Dummy                0.758 (+/- 0.000)
logistic regression  0.851 (+/- 0.001)
decision tree        1.000 (+/- 0.000)
random forest        1.000 (+/- 0.000)
XGBoost              0.909 (+/- 0.002)
LightGBM             0.892 (+/- 0.000)
CatBoost             0.900 (+/- 0.001)
Voting                             NaN
Voting_ndt           0.921 (+/- 0.001)
```

Still the results are not better than the best performing model.

1.6.1 Why combine estimators?

- It didn't happen here but how could the average do better than the best model???
  - From the perspective of the best estimator (in this case CatBoost), why are you adding on worse estimators??

Here's how this can work:

Example	log reg	rand forest	cat boost	Averaged model
1				=>
2				=>
3				=>

In short, as long as the different models make **different mistakes**, this can work.

1.6.2 Why not always do this?

1. `fit`/`predict` **time**.
2. Reduction in **interpretability**.
3. Reduction in code **maintainability** (e.g. Netflix prize).

1.6.3 What kind of estimators can we combine?

- You can **combine**
  - completely different estimators, or similar estimators.
  - estimators trained on **different samples**.
  - estimators with **different hyperparameter** values.

1.7 Stacking

- Another **type of ensemble** is stacking.
- Instead of averaging the outputs of each estimator, use their **outputs as inputs** to another ***model***.
- By default for classification, it uses **logistic regression**.
  - We don't need a complex model here necessarily, more of a weighted average.
  - The features going into the logistic regression are the classifier outputs, *not* the original features!
  - So the number of coefficients = the number of base estimators!

[41]:
```
from sklearn.ensemble import StackingClassifier
```

The code starts to get **too slow** here; so we'll remove CatBoost.

[42]:
```
classifiers_nocat = classifiers.copy()
del classifiers_nocat["CatBoost"]
```

[43]:
```
stacking_model = StackingClassifier(list(classifiers_nocat.items()))
```

[44]:
```
stacking_model.fit(X_train, y_train);
```

What's going on in here?

- It is doing cross-validation by itself by default (see documentation)
  - It is fitting the base estimators on the training fold
  - And the predicting on the validation fold
  - And then fitting the meta-estimator on that output (on the validation fold)

  Note that `estimators_` are fitted on the full X while `final_estimator_` is trained using cross-validated predictions of the base estimators using `cross_val_predict`.

Here is the input features (X) to the meta-model:

[45]:
```
X_valid_sample = X_train.sample(4, random_state=2)
y_valid_sample = y_train[X_valid_sample.index]
```

[46]:
```
stacking = {
    name: pipe.predict_proba(X_valid_sample)[:, 1]
    for (name, pipe) in stacking_model.named_estimators_.items()
}
stacking.update(y_valid_sample.to_frame().to_dict('list'))
pd.DataFrame(stacking)
```

[46]:
```
   logistic regression  decision tree  random forest   XGBoost  LightGBM  \
0             0.566252            0.0           0.12  0.249272  0.433701
1             0.000982            0.0           0.00  0.006137  0.006493
2             0.140072            0.0           0.05  0.046355  0.080048
3             0.004713            0.0           0.00  0.002433  0.003702

  income
```

```
0   <=50K
1   <=50K
2   <=50K
3   <=50K
```

- Our meta-model is logistic regression (which it is by default).
- Let's look at the learned coefficients.

[47]:
```
pd.DataFrame(
    data=stacking_model.final_estimator_.coef_[0],
    index=classifiers_nocat.keys(),
    columns=["Coefficient"],
)
```

[47]:
```
                     Coefficient
logistic regression     0.763134
decision tree          -0.011344
random forest           0.219009
XGBoost                 2.022948
LightGBM                3.684280
```

[48]:
```
stacking_model.final_estimator_.intercept_
```

[48]:
```
array([-3.31967882])
```

- It seems that the LightGBM is being trusted the most.

[49]:
```
stacking_model.predict(test_g50k)
```

[49]:
```
array(['>50K', '>50K', '>50K', '<=50K'], dtype=object)
```

[50]:
```
stacking_model.predict_proba(test_g50k)[:,1]
```

[50]:
```
array([0.96604036, 0.78658385, 0.77137857, 0.11803313])
```

(This is the `predict_proba` from logistic regression)

Let's see how well this model performs.

[51]:
```
results["Stacking_nocat"] = mean_std_cross_val_scores(
    stacking_model, X_train, y_train, return_train_score=True,␣
 ↪scoring=scoring_metric
)
```

[52]:
```
pd.DataFrame(results).T
```

[52]:
```
                                 fit_time          score_time          test_score  \
Dummy                  0.016 (+/- 0.006)  0.012 (+/- 0.004)  0.758 (+/- 0.000)
logistic regression    2.360 (+/- 0.338)  0.044 (+/- 0.016)  0.850 (+/- 0.006)
decision tree          0.343 (+/- 0.031)  0.031 (+/- 0.002)  0.813 (+/- 0.003)
```

```
random forest         2.409 (+/- 0.367)  0.151 (+/- 0.011)  0.857 (+/- 0.004)
XGBoost               6.625 (+/- 6.009)  0.079 (+/- 0.007)  0.870 (+/- 0.003)
LightGBM              0.376 (+/- 0.047)  0.074 (+/- 0.006)  0.871 (+/- 0.004)
CatBoost              8.199 (+/- 0.663)  0.215 (+/- 0.035)  0.872 (+/- 0.003)
Voting               20.037 (+/- 3.541)  0.554 (+/- 0.019)  0.868 (+/- 0.003)
Voting_ndt           18.040 (+/- 3.314)  0.611 (+/- 0.053)  0.872 (+/- 0.003)
Stacking_nocat       56.018 (+/- 8.566)  0.399 (+/- 0.052)  0.872 (+/- 0.004)

                           train_score
Dummy                0.758 (+/- 0.000)
logistic regression  0.851 (+/- 0.001)
decision tree        1.000 (+/- 0.000)
random forest        1.000 (+/- 0.000)
XGBoost              0.909 (+/- 0.002)
LightGBM             0.892 (+/- 0.000)
CatBoost             0.900 (+/- 0.001)
Voting                             NaN
Voting_ndt           0.921 (+/- 0.001)
Stacking_nocat       0.900 (+/- 0.007)
```

- The situation here is a bit mind-boggling.
- On each fold of cross-validation it is doing cross-validation.
- This is really loops within loops within loops within loops…

- We can also try a different final estimator:
- Let's `DecisionTreeClassifier` as a final estimator.

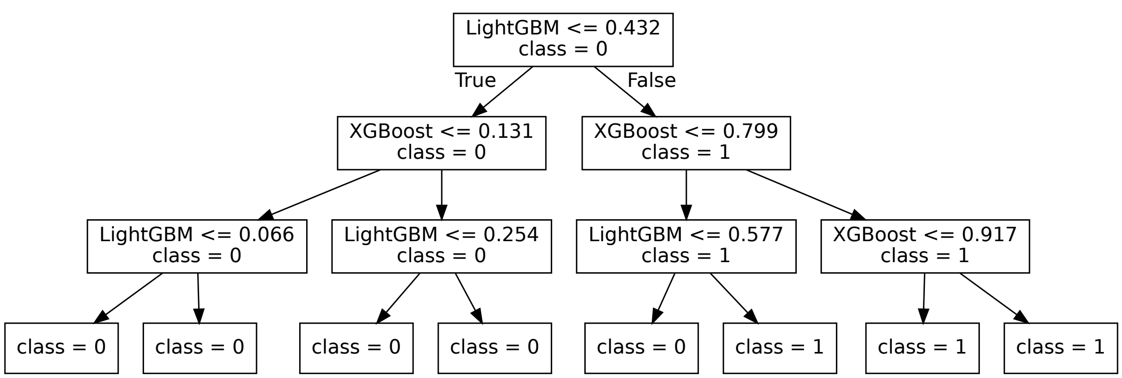
[53]:
```
stacking_model_tree = StackingClassifier(
    list(classifiers_nocat.items()),
final_estimator=DecisionTreeClassifier(max_depth=3)
)
```

The results are not very good. But we can look at the tree:

[54]:
```
stacking_model_tree.fit(X_train, y_train);
```

[55]:
```
display_tree(list(classifiers_nocat.keys()), stacking_model_tree.
final_estimator_)
```

[55]:



An effective strategy

- **Randomly generate** a bunch of models with **different hyperparameter** configurations,
- and then **stack all the models**.

Advantage and Disadvantage

- What is an advantage of ensembling multiple models as opposed to just choosing one of them?
  - You may get a **better score**.
- What is an disadvantage of ensembling multiple models as opposed to just choosing one of them?
  - **Slower**, more **code maintenance** issues.

1.8 Summary

- You have a number of models in your toolbox now.

- Ensembles are usually pretty effective.
  - Tree-based classifiers are particularly popular and effective on a wide range of problems.
  - But they trade off code complexity and speed for prediction accuracy.
  - Don't forget that hyperparameter optimization multiplies the slowness of the code!
- Stacking is a bit slower than voting, but generally higher accuracy.
  - As a bonus, you get to see the coefficients for each base classifier.
- All the above models have equivalent regression models.

Relevant papers

- Fernandez-Delgado et al. 2014 compared 179 classifiers on 121 datasets:
  - First best class of methods was Random Forest and second best class of methods was (RBF) SVMs.
- If you like to read original papers here is the original paper on Random Forests by Leo Breiman.

1.9 True or False questions on Random Forests (Class discussion)

1. Every tree in a random forest uses a different bootstrap sample of the training set. **True**
2. To train a tree in a random forest, we first randomly select a subset of features. The tree is then restricted to only using those features. **TRUE**
3. A reasonable implementation of `predict_proba` for random forests would be for each tree to "vote" and then normalize these vote counts into probabilities. **TRUE**
4. Increasing the hyperparameter max_features (the number of features to consider for a split) makes the model more complex and moves the fundamental tradeoff toward lower training error. **TRUE**
5. A random forest with only one tree is likely to get a higher training error than a decision tree of the same depth. **FALSE**

How would you carry out “soft voting” with `predict_proba` output instead of hard voting for random forests?

```
[ ]:
```