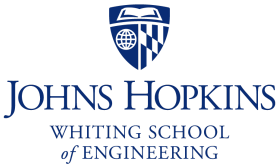
Ensemble Learning



Introduction

Ensemble methods combine multiple hypotheses, aiming to
form a better one than the best hypothesis alone. Often, ensemble methods use multiple weak learners to build a strong learner.

A **weak learner** is one that consistently generates better predictions than random.

Ensemble methods use multiple learning algorithms (or instances) to obtain better predictive performance than could be obtained from any of the individual learning algorithms (or instances) alone. A machine learning ensemble can consist of a finite set of alternative models, which are more general classification models discretely.

The common ensemble training scheme employs numerous *weaker* classifiers that use only subsets of the dataset in terms of rows (data points) or columns (features), and then conducts a majority voting among their resulting predictions to decide on a particular single data point classification.

One important advantage of ensemble methods is the reduction of over-fitting.

Curse of Dimensionality

In machine learning one of the biggest difficulty is dealing with high number of dataset features or lack of enough data points to cover them. As an example, when a naive model requires 5 data points on each feature of a 100 feature dataset, the **joint** density function would ideally necessitate 5^{100} data points, so that the density function could be computed perfectly. This is not only too big of a dataset size, but also it might be impossible to acquire such a dataset. In practice, the curse of dimensionality is avoided by assuming the features are independent, such as the Naive Bayes assumption. The alternative approach is using ensembles of weak classifiers and bagging, so that each classifier would work on a subset of features.

Note that if the features are (asumed to be) independent one would require $5 \cdot 100$ data points to represent the dataset conforming to the same criteria above.

Bagging or Bootstrap Aggregating

Bagging trains each model in the ensemble using a randomly drawn subset of the training set. Then lets each model in the ensemble vote with equal weight. The random forest algorithm combines random decision trees with bagging to achieve high classification

accuracy while preserving model generality, which is one of the reasons why the Random Forest classifier is very popular.

Boosting

Boosting trains several weak learning algorithms and combine (i.e. summation) their weighted results. Boosting builds an ensemble by training each new model instance in such a way to improve upon the previous models mis-classify. The most common implementation of boosting is Adaboost where the classifier is composed of T many classifiers such that $F_T(x) = \sum_{t=1}^{T} f_t(x)$. The training is done such that the error is minimized at every iteration, $E_T = \sum_i E\big(F_{t-1}(x_i) + \alpha_t h(x_i)\big)$

Decision Tree Learning

Introduction

A decision tree is a layout of various outcomes associated with a series of choices related to each other. They can be utilized to weight their actions based on multiple factors such as benefits, probabilities, and costs. In machine learning the outcomes could be class predictions or new nodes that can result in class predictions eventually.

Most decision tree building algorithms split nodes into two via a comparison operator, such as a binary tree. Training a decision tree can be costly ($\mathcal{O}(n^k)$ complexity) since the problem is non-convex and a greedy algorithm has to be employed. However, using a DT model is very fast such as traversing the tree until the final leaf, class prediction is reached ($\mathcal{O}(\log n)$ complexity).

Method

Given: Collection of examples $(x, f(x))$
Return: a function h (hypothesis) that approximates f, where h is a decision tree

Note that $f(x)$ is not generally known and a dataset (of observations or records) is considered. In our common problems, we have X as the dataset and y as the observations (ground truth labels).

Input: An object or situation described by a set of attributes (or features), i.e. the dataset X
Output: a "decision" $\longrightarrow$ predicts the output value for the input, i.e. y

For a decision tree learner, the input attributes can be nominal (discrete) or numerical (continuous).

A decision tree is a tree with two types of nodes

1. Decision nodes, specifies a test on a feature (attribute) with 2 or more alternatives

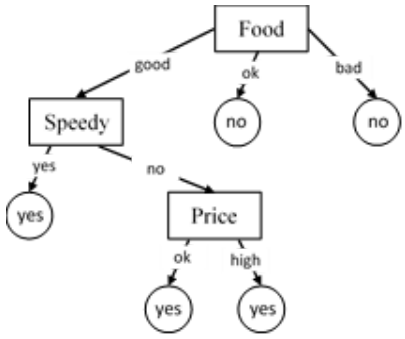
- Every decision node is in the path to a leaf node

2. Leaf nodes, indicates a classification of a data point (example, test)

Example

Food	Speedy	Price	Will Tip
good	yes	ok	yes
good	no	high	no
good	yes	high	yes
good	no	ok	yes
ok	yes	ok	no
bad	yes	ok	no
good	yes	ok	yes
good	yes	high	yes
ok	yes	ok	no
good	yes	ok	yes

Decision Tree



Given the observations (table) from 10 customers in a restaurant. According to three questions, customers are stating they tipped or not. The problem is to model this behavior in a decision tree. The following decision tree can be fit to the data. Note that this tree is **consistent** to the input data, i.e. no contradictions, non-overlapping classification, a decision surface divides the classes perfectly.

The following cells demonstrate how to use this data in a regular classifier.

In [1]:
```
import numpy as np
import pandas as pd
from sklearn.preprocessing import LabelEncoder  # Generally used for supervised

df = pd.DataFrame(
    np.array([['good', 'yes', 'ok'],
              ['good', 'no', 'high'],
              ['good', 'yes', 'high'],
              ['good', 'no', 'ok'],
              ['ok', 'yes', 'ok'],
              ['bad', 'yes', 'ok'],
              ['good', 'yes', 'ok'],
              ['good', 'yes', 'high'],
              ['ok', 'yes', 'ok'],
              ['good', 'yes', 'ok']
             ]),
    columns=['Food', 'Speedy', 'Price'])

df['Will_Tip'] = pd.Series(np.array(['yes', 'no', 'yes', 'yes', 'no', 'no', 'ye

# Convert nominals to numerical keeping the order - sklearn requirement for Dec
df['Food'] = df['Food'].replace({'bad':0, 'ok':1, 'good':2})
df['Speedy'] = df['Speedy'].replace({'no':0, 'yes':1})
df['Price'] = df['Price'].replace({'high':0, 'ok':1})

# Prepare X and y
X = df.loc[:, df.columns != 'Will_Tip'].values
y = df.loc[:, df.columns == 'Will_Tip'].values.ravel()
```

```
# Convert y values to 0, 1 for sklearn
y_le = LabelEncoder()
y = y_le.fit_transform(y)

# Sanity
df.head()
```

Out[1]:

	Food	Speedy	Price	Will_Tip
0	2	1	1	yes
1	2	0	0	no
2	2	1	0	yes
3	2	0	1	yes
4	1	1	1	no

In [2]:

```
from sklearn.tree import DecisionTreeClassifier

willtip_clf = DecisionTreeClassifier(random_state=0)
model = willtip_clf.fit(X, y)
```

The following visualization requires `graphviz` which can be installed by,
`conda install python-graphviz`

In [3]:

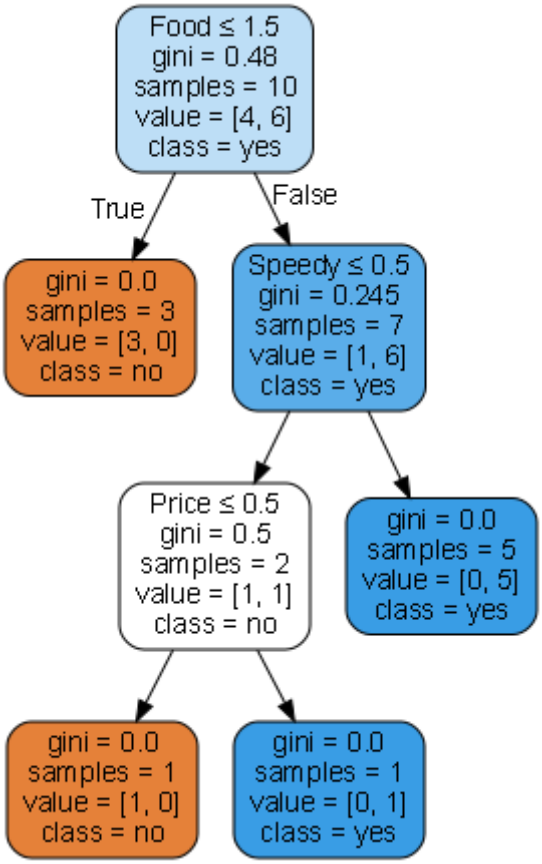
```
%matplotlib inline
import matplotlib.pyplot as plt
plt.rcParams["figure.dpi"] = 72
from IPython.display import display
from sklearn.tree import export_graphviz
from IPython.display import Image
from subprocess import run, PIPE

import graphviz
print(f'graphviz version= {graphviz.__version__}')
```

```
graphviz version= 0.20.1
```

In [4]:

```
# export_graphviz will generate a DOT image file in the folder with this notebo
dot_data = export_graphviz(willtip_clf,
                           out_file='wt_tree.dot',
                           feature_names=df.columns[:-1],
                           class_names=y_le.classes_,
                           filled=True,
                           rounded=True,
                           special_characters=True)

# Convert the DOT file to PNG
run(['dot', '-Tpng', 'wt_tree.dot', '-o', 'wt_tree.png', '-Gdpi=72'], shell=Tru

# Display the decision tree
Image(filename = 'wt_tree.png')
```

Out[4]:



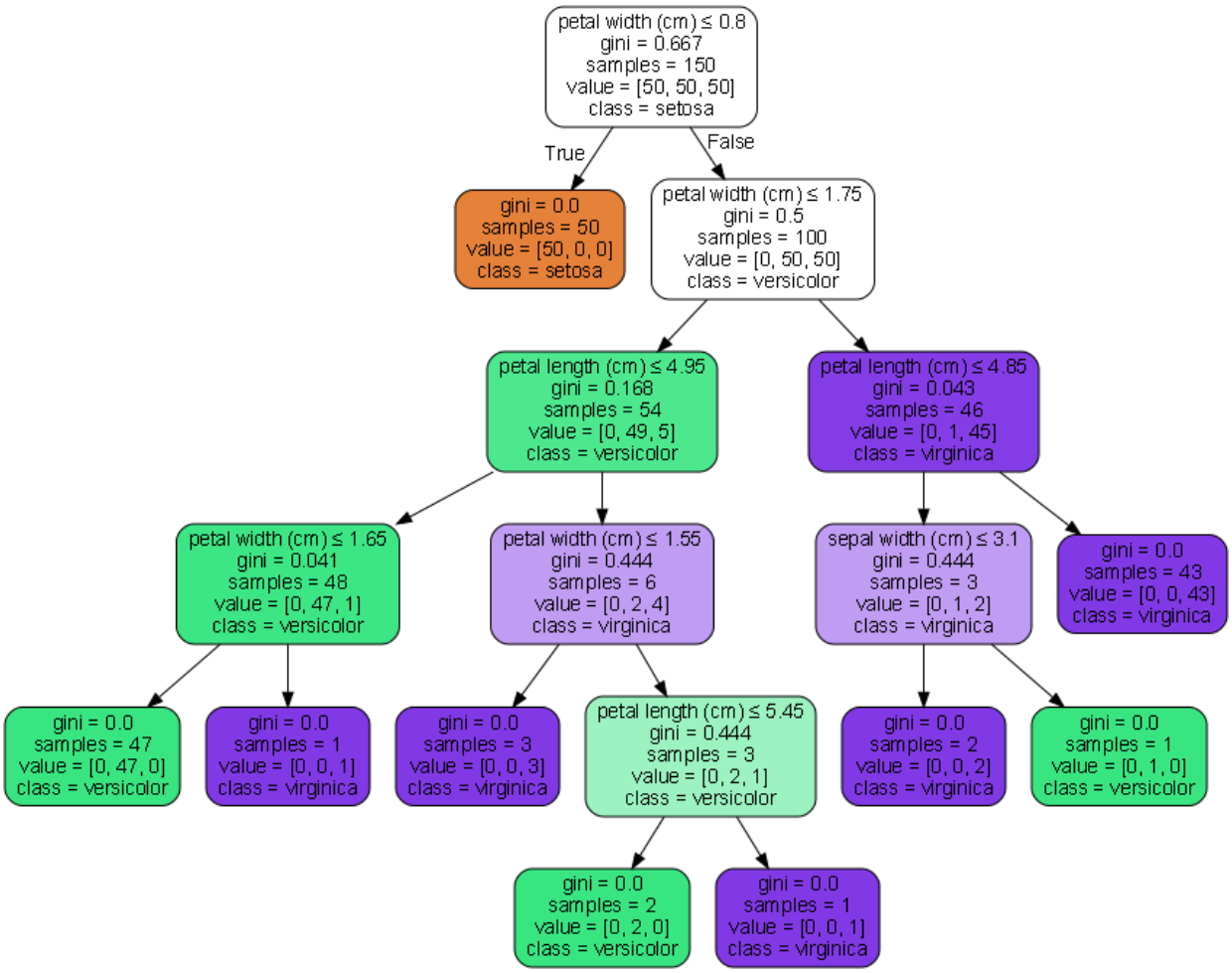
Example

Following is another decision tree example using the Iris dataset.

In [5]:

```
from sklearn import datasets

# Load the data
iris = datasets.load_iris()

iris_clf = DecisionTreeClassifier(random_state=0)
model = iris_clf.fit(iris.data, iris.target)

# Generate the learned tree using DOT language and graphviz
dot_data = export_graphviz(iris_clf,
                           out_file='iris_tree.dot',
                           feature_names=iris.feature_names,
                           class_names=iris.target_names,
                           filled=True,
                           rounded=True,
                           special_characters=True)

# Convert the DOT file to PNG
run(['dot', '-Tpng', 'iris_tree.dot', '-o', 'iris_tree.png', '-Gdpi=72'], shell

# Display the decision tree
Image(filename = 'iris_tree.png')
```

Out[5]:



ID3 Algorithm: Building a Decision Tree

Following algorithms can build decision trees.

1. Iterative Dichotomiser 3 (ID3) algorithm is used for selecting the splitting value by information gain, recursively for each level of the tree.
2. C4.5 algorithm is a modified version of the ID3 algorithm. Utilizes information gain and gain ratio for selecting the feature. It can handle numerical features and missing values.
3. CART (Classification and Regression Tree) algorithm can produce classification as well as a regression tree.

Hard Problem

If the dataset has M *binary* features then there are 2^M rows in the *complete* truth table. In addition, if the outcome, i.e. target variable, is binary then each row can have a {0, 1} output. The number of possible ways to build this truth table is 2^{2^M}. Thus, the solution space is exponential, and the decision tree algorithms are always have to be **greedy** as there is no known method to solve them in polynomial time.

All greedy algorithms have a **strategy**.

Occam's Razor

The principle of parsimony, is the principle that one should not make more assumptions than the minimum needed.

All scientific modeling and theory building is based on this principle.

Another interpretationis that all other things being equal, pick the smallest explanation.

Building a decision tree using Occam's Razor principle results in the **smallest** tree possible. The approach also minimizes overfitting.

Algorithm idea: Recursively pick the *most significant* attribute as root of the (sub)tree

- Use a top-down greedy search through the space of possible decision trees
- Greedy strategy - pick the highest/best values first

Measure of the worthiness of an attribute: Information Gain

$$\texttt{I} = -\sum_{i=1}^{K} P(y_i) \log_2 \left(P(y_i)\right)$$

$$\texttt{I}\left(\frac{p}{p+n}, \frac{n}{p+n}\right) = -\frac{p}{p+n}\log_2\left(\frac{p}{p+n}\right) - \frac{n}{p+n}\log_2\left(\frac{n}{p+n}\right)$$

$\texttt{I}$ is also called **entropy** of the prior, p and n are positive and negative class labels.

A good strategy is starting with the feature that has the highest entropy of the dependent variable.

- *Information* answers questions. The more clueless I am about a question, the more information the answer contains.

Gini Index, also known as Gini impurity, corresponds to the probability of a specific feature that is classified incorrectly when selected randomly. As an example, if all data points could be classified into one particular class then it would be called pure.

The features with the lowest Gini Index would be preferred during the tree building.

$$\text{Gini Index} = 1 - \sum_{i=1}^{C} P(i)^2$$

Intuition

Pick the attribute that reduces the entropy (uncertainity) the most.

$$\texttt{IG}(\mathrm{A}) = \texttt{I} - \mathrm{remainder}(\mathrm{A})$$

$\mathrm{remainder}(\mathrm{A})$ is the remaining uncertainity after getting info (i.e. using on a decision tree to decide, split)

$$\text{remainder(A)} = \sum_{i=1}^{K} \frac{p_i + n_i}{p + n} \mathtt{I}\left(\frac{p_i}{p + n}, \frac{n_i}{p + n}\right)$$

Gini Index vs Information Gain

- Gini Index favors bigger partitions
- Information Gain favors smaller partitions having smaller counts with multiple specific values
- CART uses Gini
- ID3 and C4.5 uses IG
- Gini is computed by correct/incorrect target values
- IG computes the difference between entropy before and after the split and indicates the impurity in classes of elements
- Pick the lowest Gini
- Pick the highest IG

Random Forest

The Random Forest classifier is a method that combines the decision trees and ensemble learning. The forest is composed of many trees that use randomly picked data features (attributes) as their input. The forest generation process constructs a collection of trees with controlled variance. The resulting prediction can be decided by majority voting or weighted voting. Random Forests have several advantages, such as a low number of control and model parameters, resistance to overfitting, no requirement for feature selection or feature reduction because they can use a large number of potential attributes. If some features are not useful for prediction, the algorithm will simply ignore them. One important advantage of Random Forest is that the variance of the model decreases as the number of trees in the forest increase, whereas the bias remains the same.

Random Forests have some disadvantages such as low model interpretability, performance loss due to correlated (dependent) variables, and dependence on the random number generator of the implementation. Note that all these disadvantages (except for the third one which is relatively easy to fix) are inherent to several other popular classifiers.

Random Forest Implementation Details

Number of features in each tree can be set to either $\sqrt{M}$, or $\log_2 (M + 1)$ for a dataset $X \in \mathbb{R}^{N \times M}$

Ensemble size determination remains to be a challenge. In practice a size is picked subjectively according to the number of features and data points.

Demonstrating the Power of Numerous Weak Classifiers as a Team

Note that not all datasets would benefit an ensemble classifier. The following cells demonstrate the power of the ensemble classifier with the following steps.

- Data exploration
- Evaluating regular classifiers on the dataset
- Building an ensemble

Recall that a few strategies to build a weak learner:

- Use a subset of dataset in training
- Use a subset of features in training
- Use a weak learner (generally more primitive and have higher generalization, or abstraction)

Note that the test on the weak learner has to match the subset features that the learning model uses.

In [6]:
```
# Locate and load the data file
df = pd.read_csv('../../../EP_datasets/titanic_preprocessed.csv')

# Sanity check
print(f'#rows={len(df)}, #columns={len(df.columns)}')
df.head()
```

```
#rows=891, #columns=69
```

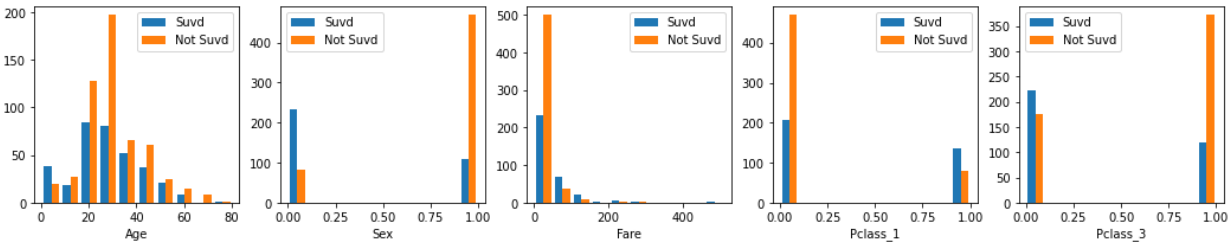
Out[6]:

	Sex	Age	SibSp	Parch	Fare	Title_Master	Title_Miss	Title_Mr	Title_Mrs	Title_Officer
0	1	22.0	1	0	7.2500	0	0	1	0	0
1	0	38.0	1	0	71.2833	0	0	0	1	0
2	0	26.0	0	0	7.9250	0	1	0	0	0
3	0	35.0	1	0	53.1000	0	0	0	1	0
4	1	35.0	0	0	8.0500	0	0	1	0	0

5 rows × 69 columns

In [7]:
```
# Data exploration, a few plots
def plt_var(_col):
    plt.hist([df[df['Survived']==1][_col],df[df['Survived']==0][_col]], label=[
    plt.xlabel(_col)
    plt.legend()

plt.figure(figsize=(18, 3), dpi=72)
plt.subplot(1, 5, 1)
plt_var('Age')

plt.subplot(1, 5, 2)
```

```
plt_var('Sex')

plt.subplot(1, 5, 3)
plt_var('Fare')

plt.subplot(1, 5, 4)
plt_var('Pclass_1')

plt.subplot(1, 5, 5)
plt_var('Pclass_3')

plt.show()
```



Evaluate numerous classifiers by a stratified 10-fold CV as in the following cells.

In [8]:
```
# Prepare the input X and y
dfX = df.loc[:, df.columns != 'Survived']
dfy = df.loc[:, df.columns == 'Survived'].values.ravel()

# Sanity check
print(f'N={len(dfX)}, M={len(dfX.columns)}')
```

```
N=891, M=68
```

In [9]:
```
# Set our main data structures X and y
X = dfX.values
y = dfy
```

In [10]:
```
from sklearn.model_selection import StratifiedKFold
from sklearn.metrics import accuracy_score

# 10-fold CV evaluation of a classifier
def eval_classifier(_clf, _X, _y):
    accuracies = []
    kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=0)
    for train_index, test_index in kf.split(_X, _y):
        _clf.fit(_X[train_index], _y[train_index])
        y_pred = _clf.predict(_X[test_index])
        accuracies += [accuracy_score(_y[test_index], y_pred)]
    return np.array(accuracies)
```

In [11]:
```
from sklearn.naive_bayes import GaussianNB

acc = eval_classifier(GaussianNB(),
                      X, y)
print(f'Naive Bayes CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3f}')
```

```
Naive Bayes CV accuracy=0.45 ±0.016
```

In [12]:
```
from sklearn.svm import LinearSVC

# Problem is difficult and LinearSVC does not converge
# Increase the convergence tolerance to a very high number
acc = eval_classifier(LinearSVC(tol=10),
                      X, y)
print(f'Linear SVC CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3f}')
```

```
Linear SVC CV accuracy=0.72 ±0.137
```

In [13]:
```
from sklearn.svm import LinearSVC

# See balancing the classes with penalties help or not
acc = eval_classifier(LinearSVC(class_weight='balanced', tol=10),
                      X, y)
print(f'Linear SVC CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3f}')
```

```
Linear SVC CV accuracy=0.75 ±0.092
```

In [14]:
```
from sklearn.svm import SVC

acc = eval_classifier(SVC(kernel='rbf', gamma=.5, C=2),
                      X, y)
print(f'SVM RBF CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3f}')
```

```
SVM RBF CV accuracy=0.68 ±0.039
```

In [15]:
```
from sklearn.svm import SVC

acc = eval_classifier(SVC(kernel='rbf', gamma=.05, C=2),
                      X, y)
print(f'SVM RBF CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3f}')
```

```
SVM RBF CV accuracy=0.73 ±0.045
```

In [16]:
```
from sklearn.linear_model import LogisticRegression

acc = eval_classifier(LogisticRegression(solver='lbfgs', multi_class='auto', ma
                      X, y)
print(f'Logistic Regression CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(ac
```

```
Logistic Regression CV accuracy=0.81 ±0.040
```

In [17]:
```
from sklearn.neural_network import MLPClassifier

acc = eval_classifier(MLPClassifier(hidden_layer_sizes=(50,100),max_iter=500),
                      X, y)
print(f'Neural Network CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3
```

```
Neural Network CV accuracy=0.77 ±0.059
```

In [18]:
```
acc = eval_classifier(MLPClassifier(hidden_layer_sizes=(20,50,50,20),max_iter=2
                      X, y)
print(f'Neural Network CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3
```

```
Neural Network CV accuracy=0.78 ±0.078
```

In [19]:
```
from sklearn.neighbors import KNeighborsClassifier

acc = eval_classifier(KNeighborsClassifier(n_neighbors=1),
```

```
                        X, y)
print(f'K Nearest Neighbor CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc
```

```
K Nearest Neighbor CV accuracy=0.71 ±0.038
```

In [20]:
```
acc = eval_classifier(DecisionTreeClassifier(),
                      X, y)
print(f'Decision Tree CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3f
```

```
Decision Tree CV accuracy=0.81 ±0.037
```

In [21]:
```
from sklearn.ensemble import RandomForestClassifier

acc = eval_classifier(RandomForestClassifier(n_estimators=200, max_depth=5, ran
                      X, y)
print(f'Random Forest CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3f
```

```
Random Forest CV accuracy=0.83 ±0.041
```

Best result so far.

Ensemble

Build an ensemble using regular Naive Bayes (NB) classifier as the weak learner. As can be seen above, its performance is almost *half* of the Random Forest ensemble classifier.

The following approach uses a subset of features to generate a Naive Bayes classifier. Note that `_clf` is the weak classifier model and its `predict` function will be used. It is only called a weak model because a subset of the features is passed for learning.

Note that the first step below, has a single NB **estimator** and uses its single prediction value. On the contrary, a `RandomForestClassifier` would have `n_estimators` many decision tree classifiers, and it would use majority-vote their predictions as its prediction.

We also compute the original class `Priors` from the full dataset to be fair to the NB learner.

Note on Prediction using `predict_proba`

The ensemble scores/probabilities generated fby every model might have a large variation, therefore it is better to sum up the scores and apply `np.argmax`. For example consider the situation of an ensemble of 100 models where sixty predictions had class_0 with a P of 0.55 and forty predictions with class_1 with a P of 0.95.

In [22]:
```
# Main Priors
counts = np.unique(y, return_counts=True)
NBpriors = [counts[1][0]/len(y), counts[1][1]/len(y)]

print(NBpriors)
```

```
[0.6161616161616161, 0.3838383838383838]
```

In [23]:
```
acc = eval_classifier(GaussianNB(priors=NBpriors), X, y)
print(f'Naive Bayes CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3f}'

# For reference
acc = eval_classifier(RandomForestClassifier(n_estimators=200, max_depth=5, rar
print(f'Random Forest CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3f
```

```
Naive Bayes CV accuracy=0.45 ±0.016
Random Forest CV accuracy=0.83 ±0.041
```

In [24]:
```
# Let's build a single weak learner using GaussianNB by subset of features
def weakNB_fit(_list_cols, _X, _y):
    Xs = _X[:,_list_cols]
    return GaussianNB(priors=NBpriors).fit(Xs, _y)  # return a single NaiveBaye

def weakNB_predict(_clf, _list_cols, _X):
    Xs = _X[:,_list_cols]
    return _clf.predict(Xs), _clf.predict_proba(Xs)

# Use _m features randomly selected from _M features
# a total of n_estimators many weak learners
def features_randomsubset(_M, _m, n_estimators=1):
    from numpy.random import choice
    # returns a list of list of column choices - subset features
    return [choice(_M, _m, replace=False) for _ in range(n_estimators)]
```

In [25]:
```
%%time

def eval_singleweak(_X, _y, _niters, _nfeatures):
    accuracies = []
    for _ in range(_niters):
        # Keep the subset features (i.e. columns) the same for a 10-fold
        cols = features_randomsubset(_X.shape[1], _nfeatures, n_estimators=1)
        # 10-fold CV
        kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=0)
        for train_index, test_index in kf.split(_X, _y):
            clf = weakNB_fit(cols[0], _X[train_index], _y[train_index])
            y_pred, y_prob = weakNB_predict(clf, cols[0], _X[test_index])
            accuracies += [accuracy_score(_y[test_index], y_pred)]
    return np.array(accuracies)

# Measure individual weak learner performance
acc = eval_singleweak(X, y, 100, 5)

# Sanity
print(f'total #results= {len(acc)}')

print(f'Weak learners average Acc= {np.mean(acc):.2f} {chr(177)}{np.std(acc):.3
```

```
total #results= 1000
Weak learners average Acc= 0.50 ±0.135
CPU times: total: 1.03 s
Wall time: 1.03 s
```

Interestingly, by using the good old `GaussianNB` with a small dataset and accumulating its accuracy shows a jump in performance compared to the plain NB using all features at the same time.

Question 1: Why does the performance increase?
Question 2: Why there is an increase in the standard deviation?

Note: Attempting number of features `nfeatures`=3 or so might cause individual `GaussianNB` classifier to fail due to data problems with that particular subset of features (such as no data point for the target, or all same values for a column).

In [26]:
```
# Generate numerous trained NB classifiers as weak learners
def ensembleNB_fit(_ensemble_cols, _X, _y):
    # the list of ensemble columns have a column list for every member of the e
    n_estimators = len(_ensemble_cols)
    # list of weak learners
    ensemble_clf = []
    for j in range(n_estimators):
        ensemble_clf += [weakNB_fit(_ensemble_cols[j], _X, _y)]

    return ensemble_clf

# Using trained ensemble, predict the outcome by majority voting
def ensembleNB_predict(_ensemble_clf, _ensemble_cols, _Xtest):
    from collections import defaultdict
    n_estimators = len(_ensemble_clf)
    assert n_estimators==len(_ensemble_cols)  # Error check
    # weak learner predictions
    ypred_e, yprob_e = [], []
    for j in range(n_estimators):
        res = weakNB_predict(_ensemble_clf[j], _ensemble_cols[j], _Xtest)
        ypred_e += [res[0]]
        yprob_e += [res[1]]  # score/probability of the prediction
    # majority voting for each data point in _Xtest
    ypred = []
    for i in range(_Xtest.shape[0]):
        ypred_scores = defaultdict(float)
        for j in range(n_estimators):
            for c, p in enumerate(yprob_e[j][i]):
                # a proper score is necessary
                ypred_scores[c] += p
        ix = max(ypred_scores.items(), key=lambda a: a[1])
        ypred += [ix[0]]

    return np.array(ypred)
```

In [27]:
```
%%time

# Attemp 10-fold CV using ensemble_fit, ensemble_predict
# _nfeatures: feature subset size
def eval_ensemble(_X, _y, _niter, _n_estimators, _nfeatures):
    accuracies = []
    for i in range(_niter):
        # Keep subset features, columns same for a 10-fold
        cols = features_randomsubset(_X.shape[1], _nfeatures, n_estimators=_n_e
        # 10-fold CV
        kf = StratifiedKFold(n_splits=10, shuffle=True, random_state=i)
        for train_index, test_index in kf.split(_X, _y):
            e_clf = ensembleNB_fit(cols, _X[train_index], _y[train_index])
            y_pred = ensembleNB_predict(e_clf, cols, _X[test_index])
            accuracies += [accuracy_score(_y[test_index], y_pred)]
```

```
    return np.array(accuracies)

# Measure ensemble weak learners performance
acc = eval_ensemble(X, y, 10, 200, 7)

print(f'Ensemble learners average Acc= {np.mean(acc):.2f} {chr(177)}{np.std(acc
```

```
Ensemble learners average Acc= 0.39 ±0.012
CPU times: total: 13.9 s
Wall time: 13.9 s
```

Question: Above performance results are not very promising, so do you think we have a bug in the scripts?

Improvements

More or less half of the features are not useful in this dataset. So, subsets of them are not useful either. Random Forest classifier ignores many of these by employing **entropy** or **information gain** strategy. However, such capability does not exist for a plain NB classifier. NB classifier has to use all features that it was provided. Thus, we need to do some kind of feature ranking.

Use a simple approach and check the **correlation** between `X` columns and the target variable `y`. Drop the bottom performers.

Note: Normalization of features would help the correlation analytic

In [28]:
```
# Direct correlation between each column of X and the target y
corrs = np.array([np.correlate(X[:,j], y)[0] for j in range(X.shape[1])])

# Reverse sort, numpy array negation reverses the order
ranks = np.argsort((-corrs))

# Display top-9 and bot-5
rankings = [(f'{corrs[j]:.1f}', df.columns[j]) for j in ranks]
display(rankings[:9])
display(rankings[-5:])
```

```
[('16551.2', 'Fare'),
 ('9639.2', 'Age'),
 ('663.0', 'FamilySize'),
 ('254.0', 'Ticket_XXX'),
 ('219.0', 'Embarked_S'),
 ('206.0', 'Cabin_U'),
 ('169.0', 'SmallFamily'),
 ('162.0', 'SibSp'),
 ('159.0', 'Parch')]
[('0.0', 'Ticket_SCA4'),
 ('0.0', 'Ticket_SCA3'),
 ('0.0', 'Ticket_Fa'),
 ('0.0', 'Ticket_LP'),
 ('0.0', 'Ticket_A')]
```

In [29]:
```
# Find columns with correlation=2.0 or less and then delete them from X
delcols = [(j, f'{corrs[j]:.1f}', df.columns[j]) for j in ranks if corrs[j]<=2

print(delcols)
```

```
[(47, '2.0', 'Ticket_SCAH'), (60, '2.0', 'Ticket_SWPP'), (55, '2.0', 'Ticket_S
OTONOQ'), (50, '2.0', 'Ticket_SCParis'), (42, '2.0', 'Ticket_PP'), (33, '2.0',
'Ticket_C'), (29, '2.0', 'Ticket_A5'), (20, '2.0', 'Cabin_G'), (39, '1.0', 'Ti
cket_LINE'), (43, '1.0', 'Ticket_PPP'), (44, '1.0', 'Ticket_SC'), (62, '1.0',
'Ticket_WEP'), (61, '1.0', 'Ticket_WC'), (51, '1.0', 'Ticket_SOC'), (65, '0.
0', 'Singleton'), (28, '0.0', 'Ticket_A4'), (30, '0.0', 'Ticket_AQ3'), (59,
'0.0', 'Ticket_STONOQ'), (31, '0.0', 'Ticket_AQ4'), (32, '0.0', 'Ticket_AS'),
(56, '0.0', 'Ticket_SP'), (54, '0.0', 'Ticket_SOTONO2'), (53, '0.0', 'Ticket_S
OPP'), (52, '0.0', 'Ticket_SOP'), (35, '0.0', 'Ticket_CASOTON'), (36, '0.0',
'Ticket_FC'), (48, '0.0', 'Ticket_SCOW'), (21, '0.0', 'Cabin_T'), (46, '0.0',
'Ticket_SCA4'), (45, '0.0', 'Ticket_SCA3'), (38, '0.0', 'Ticket_Fa'), (40, '0.
0', 'Ticket_LP'), (27, '0.0', 'Ticket_A')]
```

In [30]:
```
# Column numbers to delete
dd = [d[0] for d in delcols]

# Drop those columns, axis=1
Xpp = np.delete(np.array(X, copy=True), dd, axis=1)

# Xpp is new pre-processed X

# Sanity check
print(f'{X.shape}, {Xpp.shape}')
```

```
(891, 68), (891, 35)
```

In [31]:
```
# Base classifier
acc = eval_classifier(GaussianNB(priors=NBpriors), Xpp, y)
accNB, stdevNB = np.mean(acc), np.std(acc)
print(f'Naive Bayes CV accuracy={accNB:.2f} {chr(177)}{stdevNB:.3f}')

# Reference
acc = eval_classifier(RandomForestClassifier(n_estimators=200, max_depth=5, rar
print(f'Random Forest CV accuracy={np.mean(acc):.2f} {chr(177)}{np.std(acc):.3f
```

```
Naive Bayes CV accuracy=0.75 ±0.065
Random Forest CV accuracy=0.83 ±0.042
```

Notice: The plain Naive Bayes has an improved performance now, since we are helping NB with the removal of uncorrelated features. Random Forest uses the information gain metric to do this by itself.

In [32]:
```
%%time
# Measure weak learners performance on updated X
acc = eval_singleweak(Xpp, y, 100, 5)
print(f'Weak learners average Acc= {np.mean(acc):.2f} {chr(177)}{np.std(acc):.3
```

```
Weak learners average Acc= 0.70 ±0.072
CPU times: total: 953 ms
Wall time: 949 ms
```

In [33]:
```
acc = eval_ensemble(Xpp, y, 10, 200, 11)
print(f'Ensemble learners average Acc= {np.mean(acc):.2f} {chr(177)}{np.std(acc
```

```
Ensemble learners average Acc= 0.77 ±0.043
```

In [34]:
```
# The final number of features
M = Xpp.shape[1]
```

Notice: The following experiment takes around 12 minutes on my machine to run.

In [35]:
```
%%time

# Run an experiment for a full scale of number of features
valsF, accF, stdevF = np.arange(1,M+1), [], []
for nf in valsF:
    acc = eval_ensemble(Xpp, y, 10, 200, nf)
    accF += [np.mean(acc)]
    stdevF += [np.std(acc)]
```

```
CPU times: total: 8min 54s
Wall time: 8min 54s
```

Performance Comparison

Since we are varying the feature subset size for the ensemble NB, we can do something similar for the plain NB. Let's vary the features added for classification using the ranked features and plot the 10-fold CV performance. By ranking the features we are helping the plain Naive Bayes immensely.

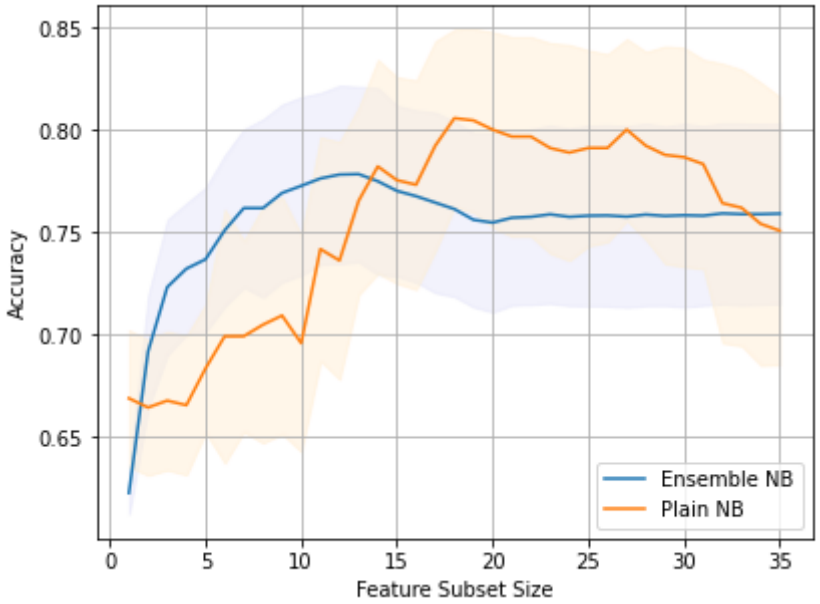
Note that both the ensemble test and plain NB test have 10 iterations to collect sufficient statistics.

In [36]:
```
%%time

# Add number of iterations - better statistics
def eval_classifier_niter(_clf, _X, _y, _niter):
    return np.array([eval_classifier(_clf, _X, _y) for _ in range(_niter)])

# Get the X matrix starting from the high-ranked variables
def getX(_X, _ncols, _ranks):
    cols = [j for n, j in enumerate(_ranks) if n < _ncols]
    return np.array(_X[:,cols])

# Iterate over the features, adding 1 by 1 by starting from the first (the best
valsNB, accNB, stdevNB = np.arange(1,M+1), [], []
for nc in valsNB:
    # feature order is coming from ranks
    X_nb = getX(X, nc, ranks)
    acc = eval_classifier_niter(GaussianNB(priors=NBpriors), X_nb, y, 10)
    accNB += [np.mean(acc)]
    stdevNB += [np.std(acc)]
```

```
CPU times: total: 3.02 s
Wall time: 3.01 s
```

In [37]:
```
# Ease of computation for the plotting of the error band
accF, stdevF = np.array(accF), np.array(stdevF)
accNB, stdevNB = np.array(accNB), np.array(stdevNB)

# Plot
```

```
plt.plot(valsF, accF, label='Ensemble NB')
plt.plot(valsNB, accNB, label='Plain NB')
plt.fill_between(valsF, accF-stdevF, accF+stdevF, color='lavender', alpha=0.5)
plt.fill_between(valsNB, accNB-stdevNB, accNB+stdevNB, color='papayawhip', alph
plt.xlabel('Feature Subset Size')
plt.ylabel('Accuracy')
plt.legend(loc='lower right')
plt.grid()
plt.show()
```



Conclusions

1. Ensemble performance is better than the plain NB when all features are added
2. Ensemble has better generalization performance (why?)
3. Ensemble performance has smaller error variance
4. Ensemble uses all features randomly picked
5. Since we ranked the features, plain NB starts adding the best features first, thus even with 1 feature its performance is 67%
6. There is an optimum number of features for the NB for this dataset
7. The plain NB performance sensitivity to the features included is high (why?)
8. The ensemble performance sensitivity to the features included is lower (why?)

Question: From the above plot, the performance of the ensemble looks a bit less than the plain NB, do you agree with this statement? Why?

References

1. Breiman, Leo. "Random forests." Machine learning 45.1 (2001): 5-32.

Exercises

Exercise 1. Add the data points `['good', 'yes', 'ok', 'no']` to the data and rebuild the `Will_Tip` tree. Observe the difference between the previous and the current.

Exercise 2. Using above dataset compare the `AdaBoostClassifier` performance to the `RandomForestClassifier` performance.

Exercise 3. Normalize features so the ranking would be more proper for the ensemble. Report improvements.

In [38]:
```
%%html
<style>
    table {margin-left: 0 !important;}
</style>
<!-- Display markdown tables left oriented in this notebook. -->
```
