CS167: Machine Learning

Random Forest

Tuesday, March 19th, 2024

Announcements

- Notebook #4: scikit library with wine quality
  - Due tonight 03/19 by 11:59pm
  - To submit, download the `ipynb` file from Colab

Today's Agenda

- Warm-Up Exercise
- Random Forest
- Random Forest Implementation using sklearn

Review of the Scikit Learn 'Algorithm'

- When working in Scikit Learn (`sklearn`), there is a general pattern that we can follow to implement any supported machine learning algorithm. It goes like this:
  - Load your data using `pd.read_csv()`
  - Split your data `train_test_split()`
  - Create your classifier/regressor object
  - Call `fit()` to train your model
  - Call `predict()` to get predictions
  - Call a metric function to measure the performance of your model

Warm-up Exercise

- Make a copy of today's Notebook
  - Day12_Random_Forests
- Make sure you change the path to match your Google Drive location
  - Load the breast-cancer-wisconsin-data.csv file

```
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import confusion_matrix
from sklearn.metrics import ConfusionMatrixDisplay
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import pandas as pd

target = "diagnosis"
predictors = data.columns.drop(target) #gets all of the columns except the target

train_data, test_data, train_sln, test_sln = train_test_split(data[predictors], data[target], test_size = 0.2, random_state=41)
print('Target column unique values: ', data[target].unique()) # M: Malignant and B: Benign
```

Warm-up Exercise

- Finish the rest

```
[ ]  # Step 1: Build a Decision Tree

     # Step 2: Normalize your data

     # Step 3: Try using a decision tree on your normalized data

     # Step 4: Print out the confusion matrix for your model
```

Today's Agenda

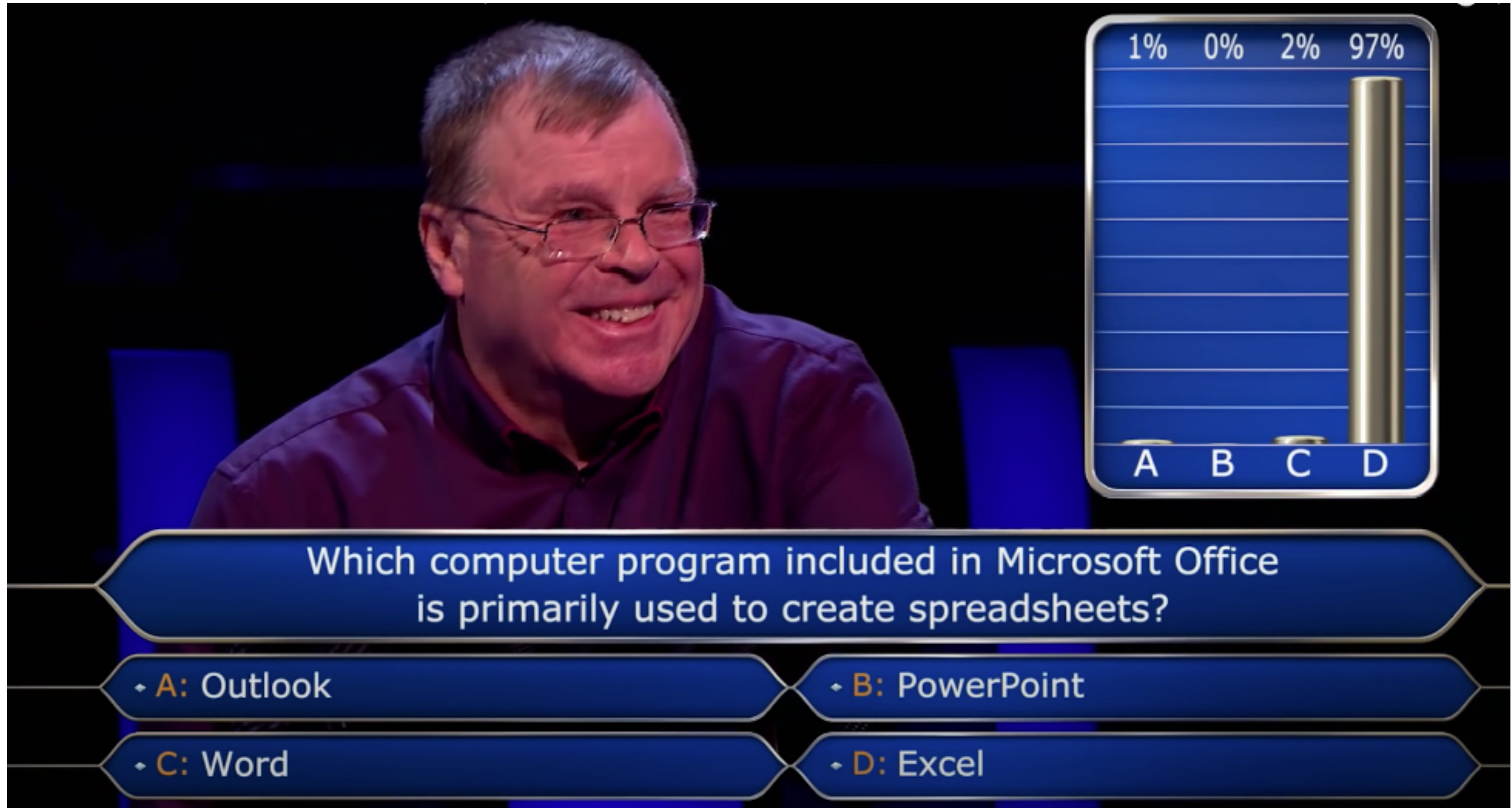
- Warm-Up Exercise
- Random Forest
- Random Forest Implementation using sklearn

Ensemble Learning

- The 'ask the audience' models:

Ensemble Learning

- The 'ask the audience' models:

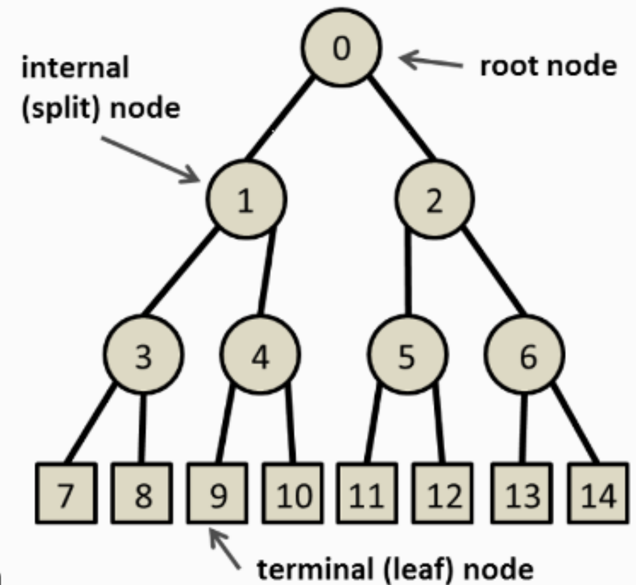


Ensemble Learning

- **Ensemble Learning:**
  - using multiple learners/hypotheses for coming up with predictions - often performs better than using one algorithm alone
  - Like crowdsourcing different machine learning models to come up with a consensus

Forest Data Structure

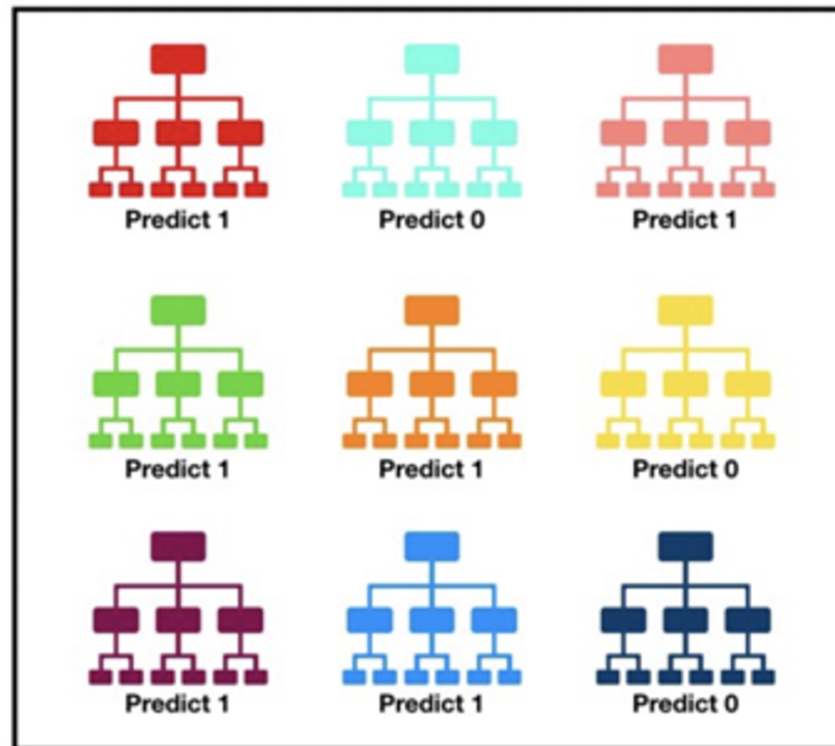
- **Tree:** a common data structure that simulates a hierarchical tree structure, with a root value and subtrees of children with a parent node, represented as a set of liked nodes.



- **Forest:** is a collection of trees

Random Forest

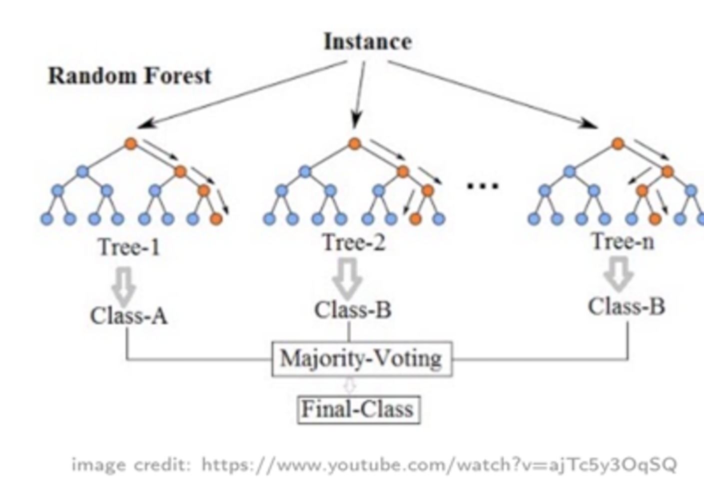
- **Big Idea:** A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models.



Visualization of a Random Forest Model Making a Prediction

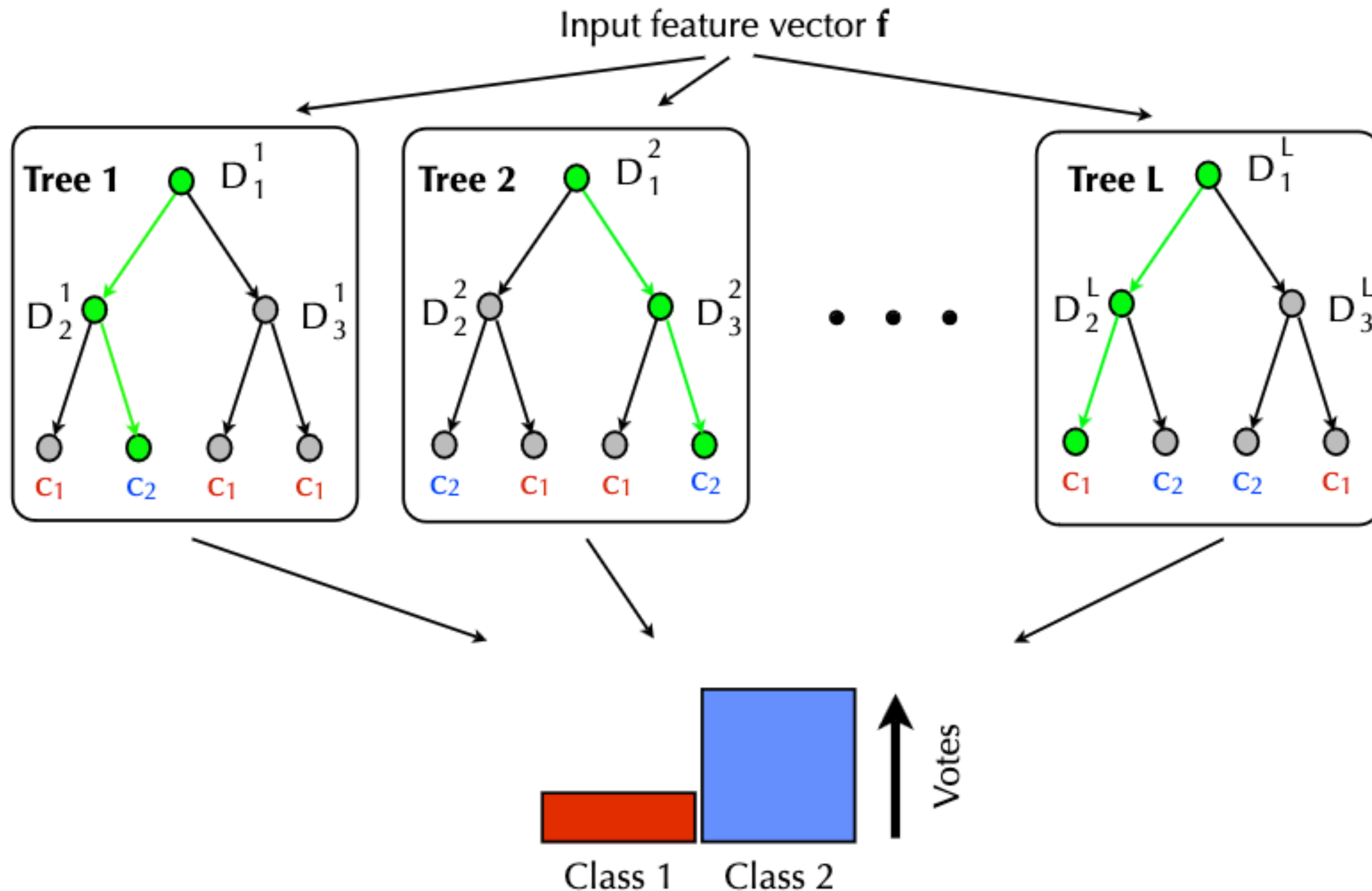
Random Forest

- **Random Forest is** an effective learning algorithm that uses an ensemble of decision trees
- **Basic idea:** build a bunch of decision trees and have them vote on the prediction



Random Forest

- **Basic idea:** build a bunch of decision trees and have them vote on the prediction



How to create different trees?

- **The power of a diverse portfolio:**
  - Just like how we want investments (e.g. stock portfolios) to be diverse…
  - Low correlation amongst investments cause stability and reliability
    - Don’t put all of your money in one industry

- **Uncorrelated models** can produce ensemble predictions that are more accurate than any of the individual predictions
  - As long as the trees don’t consistently err in the same direction

What Random Forests Need?

- There needs to be some actual signal in our features so that models built using those features do better than random guessing
- The predictions (and therefore the errors) made by the individual trees need to have low correlations with each other
  - Need to set up the trees so they all don't make the same mistakes
- How can we do this?
  - Introduce some randomness …

How to introduce Randomness?

- **Bagging:**
  - What data is used for the training sets?

Full training set	D_1 D_2 D_3 D_4 D_5 D_6 D_7 D_8 D_9 D_{10} D_{11} D_{12}
Random "bag"	D_4 D_9 D_3 D_4 D_{12} D_{10} D_{10} D_7 D_3 D_1 D_6 D_1

- **Feature subset selection:**
  - Now the trees are split?

Full feature set	f_1 f_2 f_3 f_4 f_5 f_6 f_7 f_8 f_9 f_{10}
Subset of subset	f_4 f_6 f_7 f_9 f_{10}

Sampling

- Sample with replacement:
  - allow each instance to be picked more than once

Bagging: Bootstrapping Aggregation

- Decisions trees are very sensitive to the data they are trained on
  - small changes to the training set can result in significantly different tree structures.

- **Bagging (Bootstrap Aggregation)**
  - allow each individual tree to randomly sample from the dataset *with replacement*, resulting in different trees.

Bagging: Bootstrapping Aggregation

- Notice that with bagging we are not subsetting the training data into smaller chunks
  - Rather, if we have a sample of size N, we are still feeding each tree a training set of size N (unless specified otherwise)
  - Instead of the original training data, we take a random sample of size N with replacement

Bagging: Bootstrapping Aggregation

- **Example**
  - training data was `[1, 2, 3, 4, 5, 6]`
  - then we might give one of our trees the following list `[1, 2, 2, 3, 6, 6]`
  - both lists are of length six and that 2 and 6 are both repeated in the randomly selected training data we give to our tree (because we sample with replacement)

Feature Randomness

- **Random Forests algorithm** also uses a random subset of the features for each tree
  - the size of these subsets should also be tweaked for optimal performance
  - Usually, B is the number of trees and m is the number of features in each tree

  - Classification: $m = \sqrt{total\,features}$
  - Regression: $m = \frac{total\,features}{3}$

  - These are parameters in the algorithm that need to be tuned for each dataset for optimal performance
  - **side benefit:** features that are utilized by more trees must be important - you can find out which things the learning algorithm thinks are important

Random Forest

- In our random forest, we end up with trees that are not only trained on **different sets of data** (thanks to bagging) but also use **different features** to make decisions

- A large number of relatively uncorrelated models (trees) operating as a committee will outperform any of the individual constituent models

Random Forest Application in a Computer Vision Research Paper

Image Classification using Random Forests and Ferns

Anna Bosch
Computer Vision Group
University of Girona
aboschr@eia.udg.es

Andrew Zisserman
Dept. of Engineering Science
University of Oxford
az@robots.ox.ac.uk

Xavier Muñoz
Computer Vision Group
University of Girona
xmunoz@eia.udg.es

Abstract

We explore the problem of classifying images by the object categories they contain in the case of a large number of object categories. To this end we combine three ingredients: (i) shape and appearance representations that support spatial pyramid matching over a region of interest. This generalizes the representation of Lazebnik et al [16] from an image to a region of interest (ROI), and from appearance (visual words) alone to appearance and local shape (edge distributions). (ii) automatic selection of the regions of interest in training. This provides a method of inhibiting background clutter and adding invariance to the object instance's position, and (iii) the use of random forests (and random ferns) as a multi-way classifier. The advantage of such classifiers (over multi-way SVM for example) is the ease of training and testing.

Results are reported for classification of the Caltech-101 and Caltech-256 data sets. We compare the performance of the random forest/ferns classifier with a benchmark multi-way SVM classifier. It is shown that selecting the ROI adds about 5% to the performance and, together with the other improvements, the result is about a 10% improvement over the state of the art for Caltech-256.

mid matching of Lazebnik *et al*. [16], and an improved classifier – the SVM-KNN algorithm of Zhang *et al*. [26]. In this paper we build on both of these ideas.

First the image representation: [16] argued that Caltech-101 was essentially a scene matching problem so an image based representation was suitable. Their representation added the idea of flexible scene *correspondence* to the bag-of-visual-word representations that have recently been used for image classification [8, 22, 27]. We improve on their representation in two ways. First, for training sets that are not as constrained in pose as Caltech-101 or that have significant background clutter, treating image classification as scene matching is not sufficient. Instead it is necessary to "home in" on the object instance in order to learn its visual description. To this end we automatically learn a Region Of Interest (ROI) in each of the training images in the manner of [7]. The idea is that between a subset of the training images for a particular class there will be regions with high visual similarity (the object instances). These regions can be identified from the clutter by measuring similarity using the spatial pyramid representation of [16], but here defined over a ROI rather than over the entire image. The result is that "clean" visual exemplars [3] are obtained from the pose varying and cluttered training images.

Random Forest Application in a Computer Vision Research Paper

recognition, such as PASCAL) has a significant variation in the position of the object instances within images of the same category, and also different background clutter between images (see Fig. 2). Instead of using the entire image to learn the model, an alternative is to focus on the object instance in order to learn its visual description. To this end we describe here a method of automatically learning a rectangular ROI in each of the training images. The intuition is that between a subset of the training images for a particular class there will be regions with high visual similarity (the object instances). It is a subset due to the variability in the training images – one instance may only be similar to a few others, not to all the other training images. These "corresponding" regions can be identified from the clutter by measuring their similarity using the image representation described in section 2 but here defined over a ROI rather than over the entire image.

Suppose we know the ROI r_i in image i and the subset of s other images j that have "corresponding" object instances amongst the set of training images for that class. Then we could determine the corresponding ROIs r_j of images j by optimizing the following cost function:

$$\mathcal{L}_i = \max_{\{r_j\}} \sum_{j=1}^{s} K(D(r_i), D(r_j)) \qquad (2)$$

where $D(r_i)$ and $D(r_j)$ the descriptors for the ROIs r_i and r_j respectively, and their similarity is measured using the kernel defined by (1). Here we use a descriptor formed by concatenating the PHOG and PHOW vectors. As we do not know r_i or the subset of other images we also need to search over these, i.e. over all rectangles r_i and all subsets of size s (not containing i). This is too expensive to optimize exhaustively, so we find a sub-optimal solution by alternation: for each image i, fix r_j for all other images and search over all subsets of size s and in image i search over all regions r_i. Then cycle through each image i in turn. The value for the parameter s depends on the intra-class variation and we explore its affect on performance in section 6.

In practice this sub-optimal scheme produces useful ROIs and leads to an improvement in classification performance when the model is learnt from the ROI in each training image. Fig. 2 shows examples of the learnt ROIs for a number of classes.

3.2. Random forests classifier

A random forest multi-way classifier consists of a number of trees, with each tree grown using some form of ran-

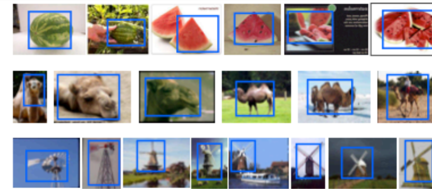


Figure 2. Automatic ROI detection. Examples from Caltech-256 for $s = 3$ for cactus, bathtub, watermelon, camel and windmill.

domization. The leaf nodes of each tree are labeled by estimates of the posterior distribution over the image classes. Each internal node contains a test that best splits the space of data to be classified. An image is classified by sending it down every tree and aggregating the reached leaf distributions. Randomness can be injected at two points during training: in subsampling the training data so that each tree is grown using a different subset; and in selecting the node tests.

Growing the trees. The trees here are binary and are constructed in a top-down manner. The binary test at each node can be chosen in one of two ways: (i) randomly, i.e. data independent; or (ii) by a greedy algorithm which picks the test that best separates the given training examples. "Best" here is measured by the information gain

$$\Delta E = -\Sigma_i \frac{|Q_i|}{|Q|} E(Q_i) \qquad (3)$$

caused by partitioning the set Q of examples into two subsets Q_i according the given test. Here $E(q)$ is the entropy $-\sum_{j=1}^{N} p_j log_2(p_j)$ with p_j the proportion of examples in q belonging to class j, and $|\ .\ |$ the size of the set. The process of selecting a test is repeated for each nonterminal node, using only the training examples falling in that node. The recursion is stopped when the node receives too few examples, or when it reaches a given depth.

Learning posteriors. Suppose that T is the set of all trees, C is the set of all classes and L is the set of all leaves for a given tree. During the training stage the posterior probabilities ($P_{t,l}(Y(I) = c)$) for each class $c \in C$ at each leaf node $l \in L$, are found for each tree $t \in T$. These probabilities are calculated as the ratio of the number of images I of class c that reach l to the total number of images that reach l. Y(I) is the class-label c for image I.

Classification. The test image is passed down each random

Today's Agenda

- Warm-Up Exercise
- Random Forest
- Random Forest Implementation using **sklearn**

Random Forest

- Start working on the class notebook. Look at the code below for decision tree

```
from sklearn import tree
from sklearn import metrics

# Let's use a single tree for comparison
# a default Decision Tree Classifier

dt = tree.DecisionTreeClassifier()
dt.fit(train_data,train_sln)
predictions = dt.predict(test_data)

print("accuracy score: ", metrics.accuracy_score(test_sln,predictions))
vals = data[target].unique() ## possible classification values (M = malignant; B = benign)
conf_mat = metrics.confusion_matrix(test_sln, predictions, labels=vals)
print(pd.DataFrame(conf_mat, index = "True " + vals, columns = "Predicted " + vals))
```

Random Forest

- Start working on the class notebook. Now, here is the code for random forest. They are very similar in structure.

```
# a Random Forest Classifier
forest = RandomForestClassifier(random_state = 0)
forest.fit(train_data,train_sln)
predictions = forest.predict(test_data)
print("accuracy score: ", metrics.accuracy_score(test_sln,predictions))

vals = data[target].unique() ## possible classification values (M = malignant; B = benign)
conf_mat = metrics.confusion_matrix(test_sln, predictions, labels=vals)
print(pd.DataFrame(conf_mat, index = "True " + vals, columns = "Predicted " + vals))
```

```
accuracy score:  0.9824561403508771
        Predicted M  Predicted B
True M           40            0
True B            2           72
```

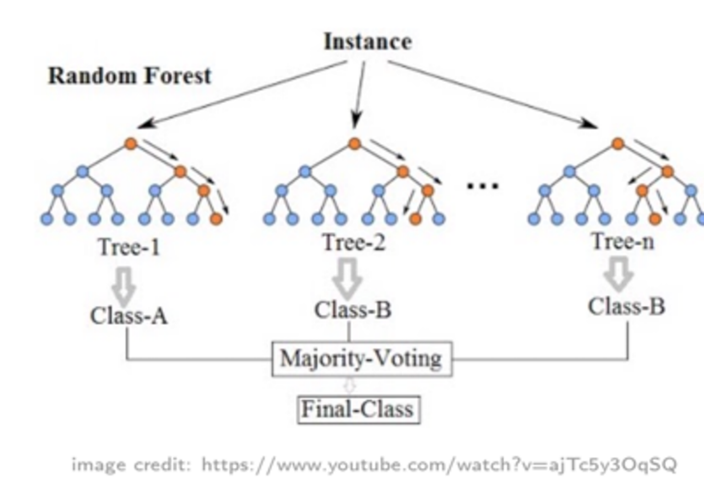
Group Exercise #1

- Look at RandomForestClassifer Documentation here
  - What is the default number of trees?
  - How does increasing or decreasing the number of trees affect accuracy?
  - What is the parameter to change to affect the number of features used?
  - How does increasing or decreasing the number of features affect accuracy?

Random Forest

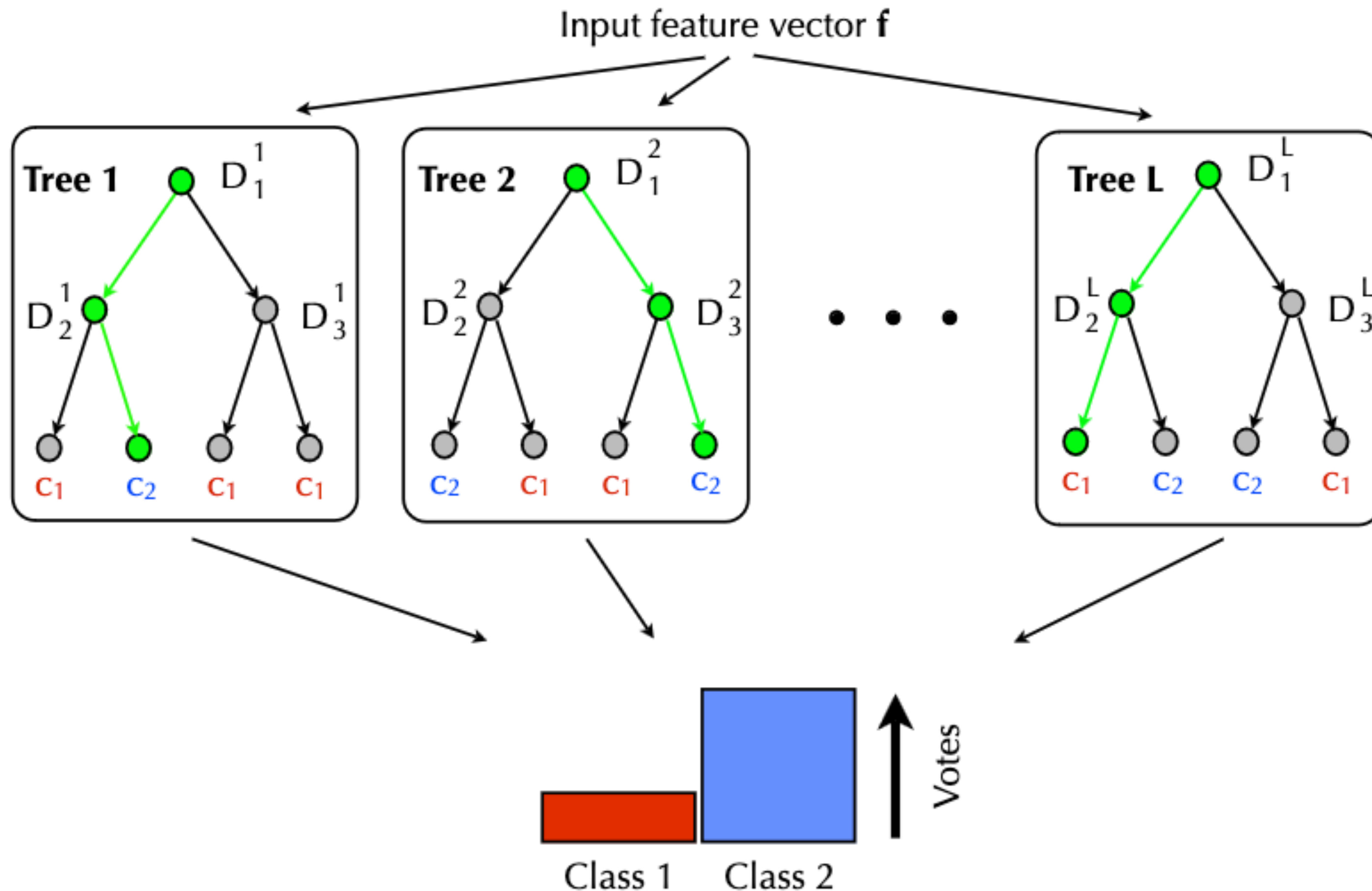
- **Random Forests** are an effective learning algorithm that uses an ensemble of decision trees

- **Basic idea:** build a bunch of decision trees and have them vote on the prediction



Random Forest

- **Basic idea:** build a bunch of decision trees and have them vote on the prediction



Feature Importance

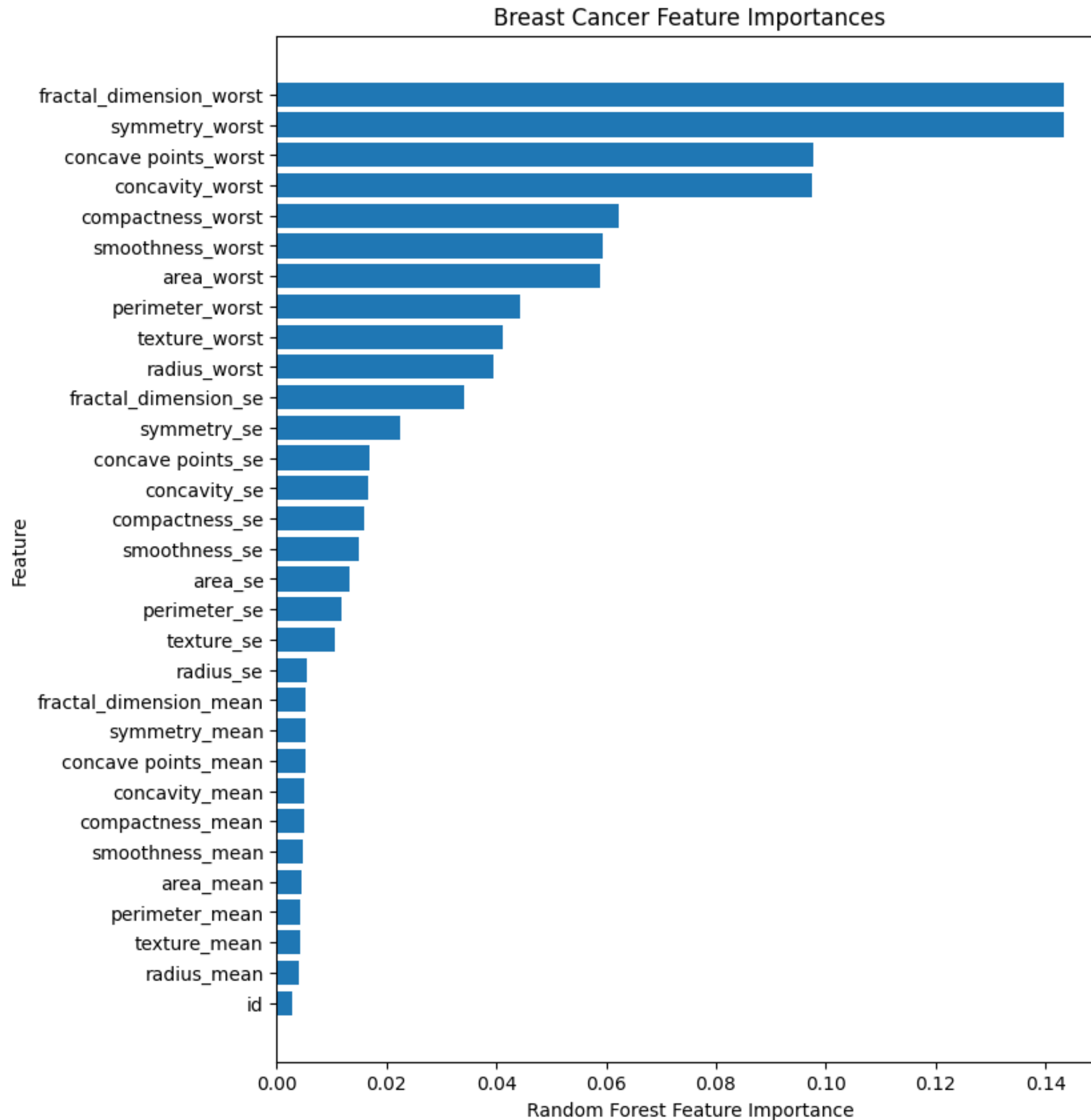
- Because we are building so many small decision trees in a random forest, we have the added benefit of being able to see what features are most commonly used as high information gain features. The code below shows how we can plot the 'Feature Importance' chart for a random forest

- In this particular run, it looks like **fractal_dimension_worst** and **symmetry_worst** were the two most important features, but there were a handful of others that were important as well

Feature Importance

```
import matplotlib.pyplot as plt
import numpy as np
%matplotlib inline

#creates a list of numbers the right size to use as the index
#and sorts the list so that the most important feature are first
index = range(len(predictors))
importances = forest.feature_importances_
sorted_indices = np.argsort(importances)

plt.figure(figsize=(8,10)) #making the table a bit bigger so the text is readable
plt.title('Breast Cancer Feature Importances')
plt.barh(range(len(sorted_indices)),importances[sorted_indices],height=0.8) #horizontal bar chart
plt.ylabel('Feature')
plt.yticks(index,predictors) #put the feature names at the y tick marks
plt.xlabel("Random Forest Feature Importance")
plt.show()
```

Feature Importance



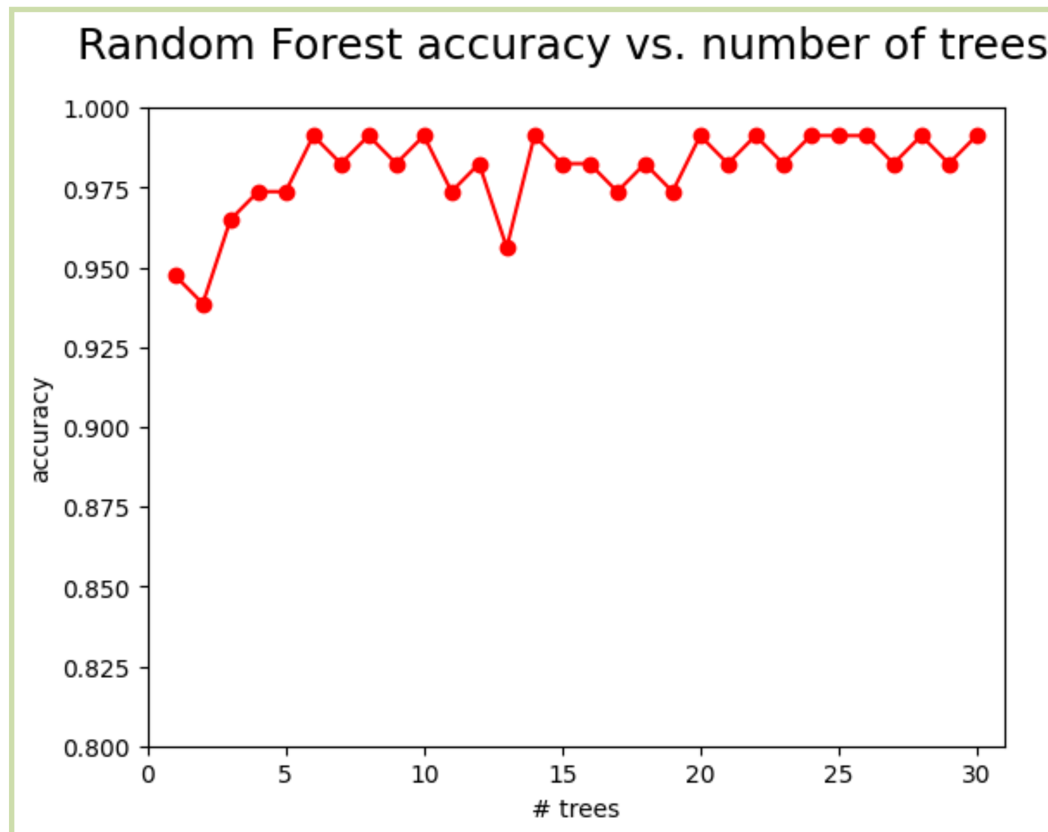
Tuning our Forest

- How can we tell how many trees to use?
- What about how many features to include in our trees?
- We can **tune** our random forest to find the best values of model parameters:

```
#This function just loops through a series of n_estimator values, builds a different model
#for each, and then plots their respective accuracies. By making it a function, it's easier
#to try out different ranges of numbers
import matplotlib.pyplot as plt

def tune_number_of_trees(n_estimator_values):
    rf_accuracies = []

    for n in n_estimator_values:

        curr_rf = RandomForestClassifier(n_estimators=n, random_state=41)
        curr_rf.fit(train_data,train_sln)
        curr_predictions = curr_rf.predict(test_data)
        curr_accuracy = metrics.accuracy_score(test_sln,curr_predictions)
        rf_accuracies.append(curr_accuracy)


    plt.suptitle('Random Forest accuracy vs. number of trees',fontsize=18)
    plt.xlabel('# trees')
    plt.ylabel('accuracy')
    plt.plot(n_estimator_values,rf_accuracies,'ro-')
    plt.axis([0,n_estimator_values[-1]+1,.8,1])

    plt.show()

tune_number_of_trees(range(1,31))
```

Tuning our Forest

- How can we tell how many trees to use?
- What about how many features to include in our trees?
- We can **tune** our random forest to find the best values of model parameters:



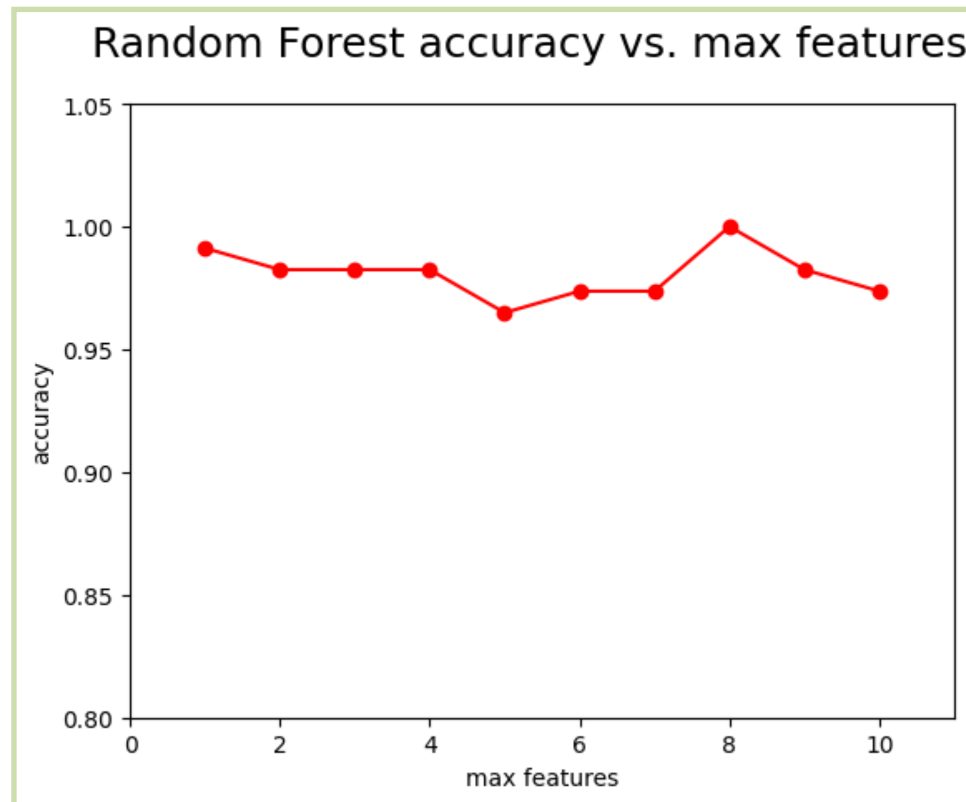
Tuning our Forest

- It looks like whether we are using small numbers of trees or large ones, the accuracy stays about the same. It appears at least sometimes that Random Forest doesn't take a lot of tuning of the number of trees.
- How can we tell how many features to be used with each tree?

```
def tune_max_features(max_features_values):
    rf_accuracies = []

    for m in max_features_values:

        curr_rf = RandomForestClassifier(n_estimators=10,max_features=m, random_state=31)
        curr_rf.fit(train_data,train_sln)
        curr_predictions = curr_rf.predict(test_data)
        curr_accuracy = metrics.accuracy_score(test_sln,curr_predictions)
        rf_accuracies.append(curr_accuracy)


    plt.suptitle('Random Forest accuracy vs. max features',fontsize=18)
    plt.xlabel('max features')
    plt.ylabel('accuracy')
    plt.plot(max_features_values,rf_accuracies,'ro-')
    plt.axis([0,max_features_values[-1]+1,.8,1.05])

    plt.show()

tune_max_features(range(1,11))
```

Tuning our Forest

- Note that the above could be subject to changes based on the initial random_state.
- For this data, which is apparently very easy to learn on (accuracy is very high), the number of features used with each tree also didn't matter much when used with an ensemble of 10 trees. This is probably something worth tuning if you have a lot of features, especially if many of them might not be very relevant.



Group Exercise #3

- Look at [RandomForestClassifer Documentation here]

  - Apply random forest to the wine dataset `winequality-white.csv` from Notebook #4
  - Can you get an R_2 score above 0.575 using RandomForestRegressor(random_state=31) (and other arguments)?

```
import pandas as pd
import numpy
from sklearn.model_selection import train_test_split

path = '/content/drive/MyDrive/cs167_fall23/datasets/winequality-white.csv' # available in the "Datasets" section of the blackboard
wines = pd.read_csv(path)
target = 'quality'
#predictors= data.columns.drop('quality')
predictors = ['fixed acidity', 'volatile acidity', 'citric acid','residual sugar', 'chlorides', 'free sulfur dioxide', 'total sulfur dioxide',
train_data, test_data, train_sln, test_sln = train_test_split(wines[predictors], wines[target], test_size = 0.2, random_state=41)
train_data.head()
```