Decision Trees and Random Forests - Machine Learning with Python

This tutorial is a part of Zero to Data Science Bootcamp by Jovian and Machine Learning with Python: Zero to GBMs

The following topics are covered in this tutorial:

- Downloading a real-world dataset
- Preparing a dataset for training
- Training and interpreting decision trees
- Training and interpreting random forests
- Overfitting, hyperparameter tuning & regularization
- Making predictions on single inputs

How to run the code

This tutorial is an executable Jupyter notebook hosted on Jovian. You can *run* this tutorial and experiment with the code examples in a couple of ways: *using free online resources* (recommended) or *on your computer*.

Option 1: Running using free online resources (1-click, recommended)

The easiest way to start executing the code is to click the **Run** button at the top of this page and select **Run on Colab**. You will be prompted to connect your Google Drive account so that this notebook can be placed into your drive for execution.

Option 2: Running on your computer locally

To run the code on your computer locally, you'll need to set up Python, download the notebook and install the required libraries. We recommend using the Conda distribution of Python. Click the **Run** button at the top of this page, select the **Run Locally** option, and follow the instructions.

Problem Statement

This tutorial takes a practical and coding-focused approach. We'll learn how to use *decision trees* and *random forests* to solve a real-world problem from Kaggle:

> **QUESTION**: The Rain in Australia dataset contains about 10 years of daily weather observations from numerous Australian weather stations. Here's a small sample from the dataset:

Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	Cloud3pm	Temp9am	Temp3pm	RainToday	RainTomorrow
2008-09-21	Melbourne	6.5	19.8	0.4	4.2	10.6	3.0	13.0	19.4	No	No
2009-07-06	Sale	4.9	13.0	0.0	2.0	6.8	6.0	8.6	11.7	No	No
2010-11-20	GoldCoast	18.8	26.4	2.0	NaN	NaN	NaN	24.0	22.1	Yes	No
2010-11-22	PearceRAAF	19.4	27.4	1.8	NaN	10.7	3.0	24.4	25.8	Yes	No
2012-04-26	Nuriootpa	5.1	16.6	0.0	1.4	1.4	7.0	12.1	15.7	No	No
2013-07-06	Sydney	7.8	17.4	0.0	4.2	9.8	0.0	10.2	17.1	No	No
2014-04-22	Perth	7.7	23.7	0.0	4.0	10.5	1.0	16.7	21.8	No	No
2014-06-08	Wollongong	11.1	16.8	0.0	NaN	NaN	1.0	14.0	15.9	No	No
2016-04-13	Sale	10.8	19.0	0.0	NaN	NaN	1.0	16.1	18.1	No	No
2017-04-11	Albany	13.0	NaN	0.0	4.0	NaN	NaN	17.8	NaN	No	NaN

As a data scientist at the Bureau of Meteorology, you are tasked with creating a fully-automated system that can use today's weather data for a given location to predict whether it will rain at the location tomorrow.

Let's install and import some required libraries before we begin.

```
#restart the kernel after installation
!pip install pandas-profiling numpy matplotlib seaborn --quiet
```

```
!pip install opendatasets scikit-learn jovian --quiet --upgrade
```

```
     |████████████████████████████████| 22.3MB 134kB/s
```

```
import opendatasets as od
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
import numpy as np
import matplotlib
import jovian
import os
```

```
%matplotlib inline

pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 150)
sns.set_style('darkgrid')
matplotlib.rcParams['font.size'] = 14
matplotlib.rcParams['figure.figsize'] = (10, 6)
matplotlib.rcParams['figure.facecolor'] = '#00000000'
```

Downloading the Data

The dataset is available at https://www.kaggle.com/jsphyg/weather-dataset-rattle-package .

We'll use the `opendatasets` library to download the data from Kaggle directly within Jupyter.

```
od.download('https://www.kaggle.com/jsphyg/weather-dataset-rattle-package')
```

```
100%|██████████| 3.83M/3.83M [00:00<00:00, 149MB/s]

Downloading weather-dataset-rattle-package.zip to ./weather-dataset-rattle-package
```

The dataset is downloaded and extracted to the folder `weather-dataset-rattle-package` .

```
os.listdir('weather-dataset-rattle-package')
```

```
['weatherAUS.csv']
```

The file `weatherAUS.csv` contains the data. Let's load it into a Pandas dataframe.

```
raw_df = pd.read_csv('weather-dataset-rattle-package/weatherAUS.csv')
```

```
raw_df
```

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindD
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0	
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	46.0	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	24.0	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	41.0	
...	...	...	...	...	...	...	...	...	...	
145455	2017-06-21	Uluru	2.8	23.4	0.0	NaN	NaN	E	31.0	

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindD
145456	2017-06-22	Uluru	3.6	25.3	0.0	NaN	NaN	NNW	22.0	
145457	2017-06-23	Uluru	5.4	26.9	0.0	NaN	NaN	N	37.0	
145458	2017-06-24	Uluru	7.8	27.0	0.0	NaN	NaN	SE	28.0	
145459	2017-06-25	Uluru	14.9	NaN	0.0	NaN	NaN	NaN	NaN	

145460 rows × 23 columns

Each row shows the measurements for a given date at a given location. The last column "RainTomorrow" contains the value to be predicted.

Let's check the column types of the dataset.

```
raw_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 145460 entries, 0 to 145459
Data columns (total 23 columns):
 #   Column         Non-Null Count   Dtype  
---  ------         --------------   -----  
 0   Date           145460 non-null  object 
 1   Location       145460 non-null  object 
 2   MinTemp        143975 non-null  float64
 3   MaxTemp        144199 non-null  float64
 4   Rainfall       142199 non-null  float64
 5   Evaporation    82670 non-null   float64
 6   Sunshine       75625 non-null   float64
 7   WindGustDir    135134 non-null  object 
 8   WindGustSpeed  135197 non-null  float64
 9   WindDir9am     134894 non-null  object 
 10  WindDir3pm     141232 non-null  object 
 11  WindSpeed9am   143693 non-null  float64
 12  WindSpeed3pm   142398 non-null  float64
 13  Humidity9am    142806 non-null  float64
 14  Humidity3pm    140953 non-null  float64
 15  Pressure9am    130395 non-null  float64
 16  Pressure3pm    130432 non-null  float64
 17  Cloud9am       89572 non-null   float64
 18  Cloud3pm       86102 non-null   float64
 19  Temp9am        143693 non-null  float64
 20  Temp3pm        141851 non-null  float64
 21  RainToday      142199 non-null  object 
 22  RainTomorrow   142193 non-null  object 
```

```
dtypes: float64(16), object(7)
memory usage: 25.5+ MB
```

Let's drop any rows where the value of the target column `RainTomorrow` in empty.

```
raw_df.dropna(subset=['RainTomorrow'], inplace=True)
```

Let's save our work before continuing.

> **EXERCISE**: Perform exploratory data analysis on the dataset and study the relationship of other columns with the `RainTomorrow` column.

```

```

```

```

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Please enter your API key ( from https://jovian.ai/ ):
API KEY: ··········
API KEY: ··········
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/aakashns/sklearn-decision-trees-random-forests
```

```
'https://jovian.ai/aakashns/sklearn-decision-trees-random-forests'
```

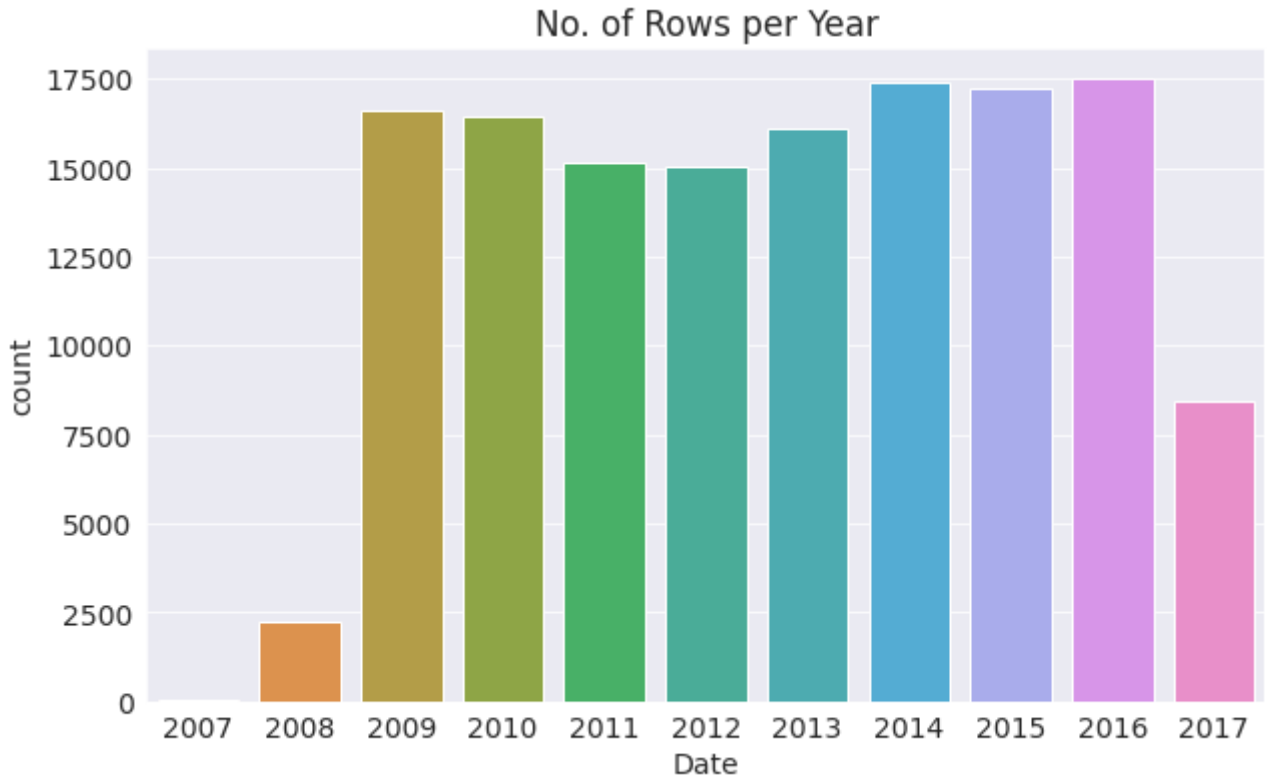
Preparing the Data for Training

We'll perform the following steps to prepare the dataset for training:

1. Create a train/test/validation split
2. Identify input and target columns
3. Identify numeric and categorical columns
4. Impute (fill) missing numeric values
5. Scale numeric values to the $(0, 1)$ range
6. Encode categorical columns to one-hot vectors

Training, Validation and Test Sets

```
plt.title('No. of Rows per Year')
sns.countplot(x=pd.to_datetime(raw_df.Date).dt.year);
```



While working with chronological data, it's often a good idea to separate the training, validation and test sets with time, so that the model is trained on data from the past and evaluated on data from the future.

We'll use the data till 2014 for the training set, data from 2015 for the validation set, and the data from 2016 & 2017 for the test set.

```
year = pd.to_datetime(raw_df.Date).dt.year

train_df = raw_df[year < 2015]
val_df = raw_df[year == 2015]
test_df = raw_df[year > 2015]
```

```
print('train_df.shape :', train_df.shape)
print('val_df.shape :', val_df.shape)
print('test_df.shape :', test_df.shape)
```

```
train_df.shape : (98988, 23)
val_df.shape : (17231, 23)
test_df.shape : (25974, 23)
```

> **EXERCISE**: Scrape the climate data for the available 14 months from http://www.bom.gov.au/climate/data and train the model using the first 12 months data, validate using the 13th-month data, and test on the 14th-month data.

```
```

Input and Target Columns

Let's identify the input and target columns.

```
input_cols = list(train_df.columns)[1:-1]
target_col = 'RainTomorrow'
```

```
train_inputs = train_df[input_cols].copy()
train_targets = train_df[target_col].copy()
```

```
val_inputs = val_df[input_cols].copy()
val_targets = val_df[target_col].copy()
```

```
test_inputs = test_df[input_cols].copy()
test_targets = test_df[target_col].copy()
```

Let's also identify the numeric and categorical columns.

```
numeric_cols = train_inputs.select_dtypes(include=np.number).columns.tolist()
categorical_cols = train_inputs.select_dtypes('object').columns.tolist()
```

```
print(numeric_cols)
```

```
['MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation', 'Sunshine', 'WindGustSpeed',
'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am', 'Humidity3pm', 'Pressure9am',
'Pressure3pm', 'Cloud9am', 'Cloud3pm', 'Temp9am', 'Temp3pm']
```

```
print(categorical_cols)
```

```
['Location', 'WindGustDir', 'WindDir9am', 'WindDir3pm', 'RainToday']
```

> **EXERCISE**: Study how various columns are correlated with the target and select just a subset of the columns, instead of all of the. Observe how it affects the results.

```

```

Imputing missing numeric values

```
from sklearn.impute import SimpleImputer
```

```
imputer = SimpleImputer(strategy = 'mean').fit(raw_df[numeric_cols])
```

```
train_inputs[numeric_cols] = imputer.transform(train_inputs[numeric_cols])
val_inputs[numeric_cols] = imputer.transform(val_inputs[numeric_cols])
test_inputs[numeric_cols] = imputer.transform(test_inputs[numeric_cols])
```

```
test_inputs[numeric_cols].isna().sum()
```

```
MinTemp          0
MaxTemp          0
Rainfall         0
Evaporation      0
Sunshine         0
WindGustSpeed    0
WindSpeed9am     0
WindSpeed3pm     0
Humidity9am      0
Humidity3pm      0
Pressure9am      0
Pressure3pm      0
Cloud9am         0
Cloud3pm         0
Temp9am          0
Temp3pm          0
dtype: int64
```

> **EXERCISE**: Try a different imputation strategy and observe how it affects the results.

Scaling Numeric Features

```
from sklearn.preprocessing import MinMaxScaler
```

```
scaler = MinMaxScaler().fit(raw_df[numeric_cols])
```

```
train_inputs[numeric_cols] = scaler.transform(train_inputs[numeric_cols])
val_inputs[numeric_cols] = scaler.transform(val_inputs[numeric_cols])
test_inputs[numeric_cols] = scaler.transform(test_inputs[numeric_cols])
```

```
val_inputs.describe().loc[['min', 'max']]
```

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9am	WindSpeed3pm	Humi
min	0.007075	0.030246	0.000000	0.000000	0.0	0.007752	0.000000	0.000000	
max	0.952830	0.948960	0.666307	0.485517	1.0	1.000000	0.669231	0.850575	

> **EXERCISE**: Try a different scaling strategy and observe how it affects the results.

```
```

Encoding Categorical Data

```
from sklearn.preprocessing import OneHotEncoder
```

```
encoder = OneHotEncoder(sparse=False, handle_unknown='ignore').fit(raw_df[categorical_c
```

```
encoded_cols = list(encoder.get_feature_names(categorical_cols))
```

```
train_inputs[encoded_cols] = encoder.transform(train_inputs[categorical_cols])
val_inputs[encoded_cols] = encoder.transform(val_inputs[categorical_cols])
test_inputs[encoded_cols] = encoder.transform(test_inputs[categorical_cols])
```

```
test_inputs
```

	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindDir9am
2498	Albury	0.681604	0.801512	0.000000	0.037723	0.525852	ENE	0.372093	NaN
2499	Albury	0.693396	0.725898	0.001078	0.037723	0.525852	SSE	0.341085	SSE
2500	Albury	0.634434	0.527410	0.005930	0.037723	0.525852	ENE	0.325581	ESE
2501	Albury	0.608491	0.538752	0.042049	0.037723	0.525852	SSE	0.255814	SE
2502	Albury	0.566038	0.523629	0.018329	0.037723	0.525852	ENE	0.193798	SE
...	...	...	...	...	...	...	...	...	...
145454	Uluru	0.283019	0.502836	0.000000	0.037723	0.525852	E	0.193798	ESE
145455	Uluru	0.266509	0.533081	0.000000	0.037723	0.525852	E	0.193798	SE
145456	Uluru	0.285377	0.568998	0.000000	0.037723	0.525852	NNW	0.124031	SE
145457	Uluru	0.327830	0.599244	0.000000	0.037723	0.525852	N	0.240310	SE
145458	Uluru	0.384434	0.601134	0.000000	0.037723	0.525852	SE	0.170543	SSE

25974 rows × 124 columns

> **EXERCISE**: Try a different encoding strategy and observe how it affects the results.

As a final step, let's drop the textual categorical columns, so that we're left with just numeric data.

```
X_train = train_inputs[numeric_cols + encoded_cols]
X_val = val_inputs[numeric_cols + encoded_cols]
X_test = test_inputs[numeric_cols + encoded_cols]
```

```
X_test
```

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9am	WindSpeed3pm	H
2498	0.681604	0.801512	0.000000	0.037723	0.525852	0.372093	0.000000	0.080460	

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9am	WindSpeed3pm	H
2499	0.693396	0.725898	0.001078	0.037723	0.525852	0.341085	0.069231	0.195402	
2500	0.634434	0.527410	0.005930	0.037723	0.525852	0.325581	0.084615	0.448276	
2501	0.608491	0.538752	0.042049	0.037723	0.525852	0.255814	0.069231	0.195402	
2502	0.566038	0.523629	0.018329	0.037723	0.525852	0.193798	0.046154	0.103448	
...	...	...	...	...	...	...	...	...	
145454	0.283019	0.502836	0.000000	0.037723	0.525852	0.193798	0.115385	0.149425	
145455	0.266509	0.533081	0.000000	0.037723	0.525852	0.193798	0.100000	0.126437	
145456	0.285377	0.568998	0.000000	0.037723	0.525852	0.124031	0.100000	0.103448	
145457	0.327830	0.599244	0.000000	0.037723	0.525852	0.240310	0.069231	0.103448	
145458	0.384434	0.601134	0.000000	0.037723	0.525852	0.170543	0.100000	0.080460	

25974 rows × 119 columns

Let's save our work before continuing.

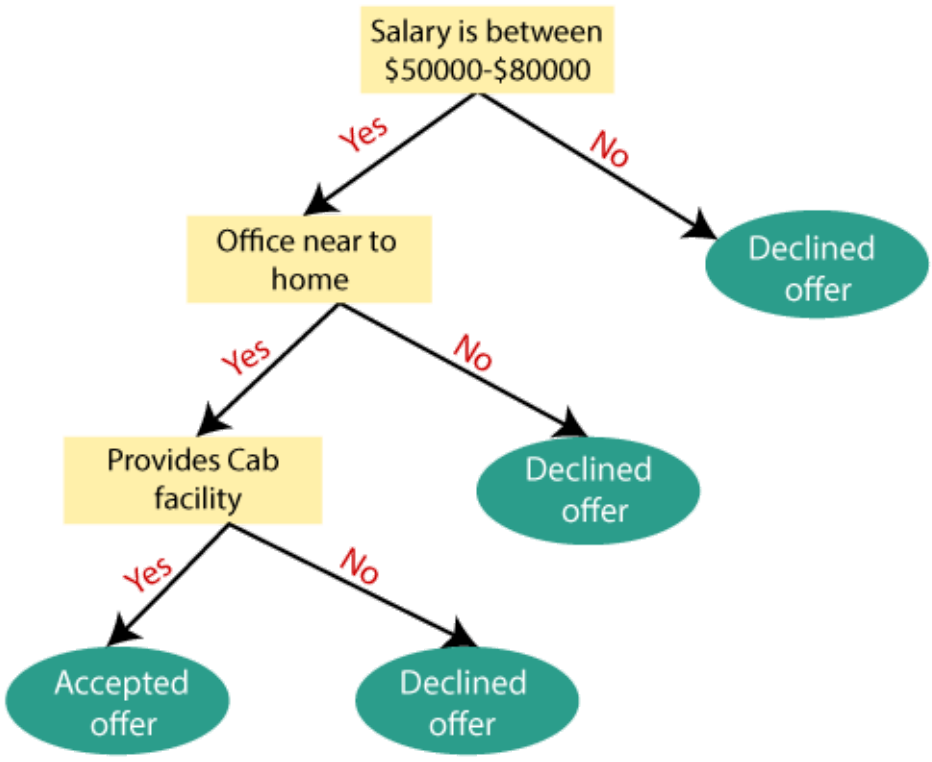
```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/aakashns/sklearn-decision-trees-random-forests
```

```
'https://jovian.ai/aakashns/sklearn-decision-trees-random-forests'
```

Training and Visualizing Decision Trees

A decision tree in general parlance represents a hierarchical series of binary decisions:



A decision tree in machine learning works in exactly the same way, and except that we let the computer figure out the optimal structure & hierarchy of decisions, instead of coming up with criteria manually.

Training

We can use `DecisionTreeClassifier` from `sklearn.tree` to train a decision tree.

```
from sklearn.tree import DecisionTreeClassifier
```

```
model = DecisionTreeClassifier(random_state=42)
```

```
%%time
model.fit(X_train, train_targets)
```

```
CPU times: user 3.31 s, sys: 2.68 ms, total: 3.31 s
Wall time: 3.31 s
```

```
DecisionTreeClassifier(random_state=42)
```

An optimal decision tree has now been created using the training data.

Evaluation

Let's evaluate the decision tree using the accuracy score.

```
from sklearn.metrics import accuracy_score, confusion_matrix
```

```
train_preds = model.predict(X_train)
```

```
train_preds
```

```
array(['No', 'No', 'No', ..., 'No', 'No', 'No'], dtype=object)
```

```
pd.value_counts(train_preds)
```

```
No     76707
Yes    22281
dtype: int64
```

The decision tree also returns probabilities for each prediction.

```
train_probs = model.predict_proba(X_train)
```

```
train_probs
```

```
array([[1., 0.],
       [1., 0.],
       [1., 0.],
```

```
       ...,
       [1., 0.],
       [1., 0.],
       [1., 0.]])
```

Seems like the decision tree is quite confident about its predictions.

Let's check the accuracy of its predictions.

```
accuracy_score(train_targets, train_preds)
```

```
0.9999797955307714
```

The training set accuracy is close to 100%! But we can't rely solely on the training set accuracy, we must evaluate the model on the validation set too.

We can make predictions and compute accuracy in one step using `model.score`

```
model.score(X_val, val_targets)
```

```
0.7921188555510418
```

Although the training accuracy is 100%, the accuracy on the validation set is just about 79%, which is only marginally better then always predicting "No".

```
val_targets.value_counts() / len(val_targets)
```

```
No     0.788289
Yes    0.211711
Name: RainTomorrow, dtype: float64
```

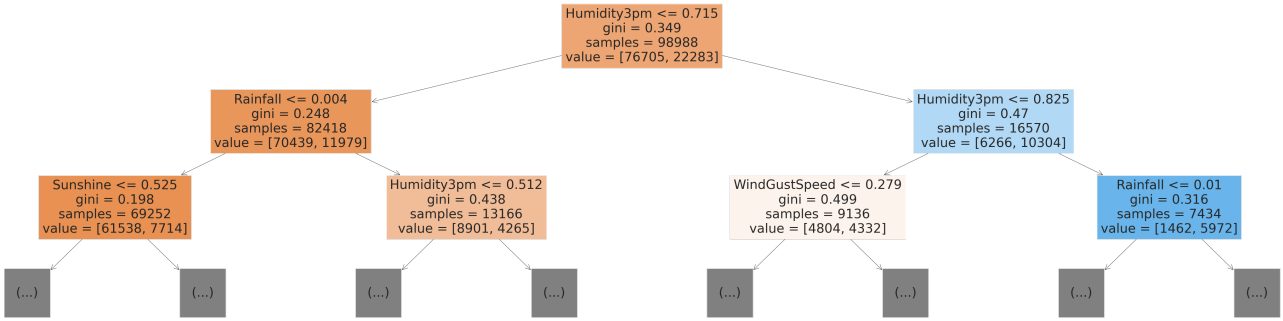
It appears that the model has learned the training examples perfect, and doesn't generalize well to previously unseen examples. This phenomenon is called "overfitting", and reducing overfitting is one of the most important parts of any machine learning project.

Visualization

We can visualize the decision tree *learned* from the training data.

```
from sklearn.tree import plot_tree, export_text
```

```
plt.figure(figsize=(80,20))
plot_tree(model, feature_names=X_train.columns, max_depth=2, filled=True);
```



Can you see how the model classifies a given input as a series of decisions? The tree is truncated here, but following any path from the root node down to a leaf will result in "Yes" or "No". Do you see how a decision tree differs from a logistic regression model?

How a Decision Tree is Created

Note the `gini` value in each box. This is the loss function used by the decision tree to decide which column should be used for splitting the data, and at what point the column should be split. A lower Gini index indicates a better split. A perfect split (only one class on each side) has a Gini index of 0.

For a mathematical discussion of the Gini Index, watch this video: https://www.youtube.com/watch?v=-W0DnxQK1Eo . It has the following formula:

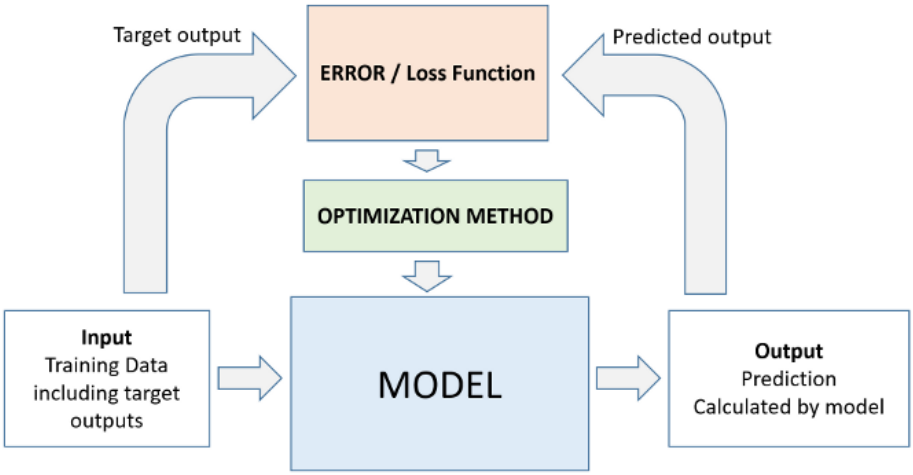
Gini Index

$$I_G = 1 - \sum_{j=1}^{c} p_j^2$$

p_j: proportion of the samples that belongs to class c for a particular node

Conceptually speaking, while training the models evaluates all possible splits across all possible columns and picks the best one. Then, it recursively performs an optimal split for the two portions. In practice, however, it's very inefficient to check all possible splits, so the model uses a heuristic (predefined strategy) combined with some randomization.

The iterative approach of the machine learning workflow in the case of a decision tree involves growing the tree layer-by-layer:



Let's check the depth of the tree that was created.

```
model.tree_.max_depth
```

48

We can also display the tree as text, which can be easier to follow for deeper trees.

```
tree_text = export_text(model, max_depth=10, feature_names=list(X_train.columns))
print(tree_text[:5000])
```

```
|--- Humidity3pm <= 0.72
|   |--- Rainfall <= 0.00
|   |   |--- Sunshine <= 0.52
|   |   |   |--- Pressure3pm <= 0.58
|   |   |   |   |--- WindGustSpeed <= 0.36
|   |   |   |   |   |--- Humidity3pm <= 0.28
|   |   |   |   |   |   |--- WindDir9am_NE <= 0.50
|   |   |   |   |   |   |   |--- Location_Watsonia <= 0.50
|   |   |   |   |   |   |   |   |--- Cloud9am <= 0.83
|   |   |   |   |   |   |   |   |   |--- WindSpeed3pm <= 0.07
|   |   |   |   |   |   |   |   |   |   |--- Pressure3pm <= 0.46
|   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |   |   |--- Pressure3pm >  0.46
|   |   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |   |   |--- WindSpeed3pm >  0.07
|   |   |   |   |   |   |   |   |   |   |--- MinTemp <= 0.32
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |   |   |--- MinTemp >  0.32
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 7
|   |   |   |   |   |   |   |   |   |--- Cloud9am >  0.83
|   |   |   |   |   |   |   |   |   |   |--- Cloud3pm <= 0.42
|   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |   |   |--- Cloud3pm >  0.42
```

```
|   |   |   |   |   |   |   |   |   |   |--- Rainfall <= 0.00
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 2
|   |   |   |   |   |   |   |   |   |   |--- Rainfall >  0.00
|   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |--- Location_Watsonia >  0.50
|   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |--- WindDir9am_NE >  0.50
|   |   |   |   |   |   |   |   |--- WindGustSpeed <= 0.25
|   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |   |--- WindGustSpeed >  0.25
|   |   |   |   |   |   |   |   |   |--- Pressure9am <= 0.54
|   |   |   |   |   |   |   |   |   |   |--- Evaporation <= 0.09
|   |   |   |   |   |   |   |   |   |   |   |--- Location_AliceSprings <= 0.50
|   |   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 4
|   |   |   |   |   |   |   |   |   |   |   |--- Location_AliceSprings >  0.50
|   |   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |   |   |--- Evaporation >  0.09
|   |   |   |   |   |   |   |   |   |   |   |--- WindGustDir_ENE <= 0.50
|   |   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |   |   |   |--- WindGustDir_ENE >  0.50
|   |   |   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |   |   |--- Pressure9am >  0.54
|   |   |   |   |   |   |   |   |   |   |--- Humidity3pm <= 0.20
|   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |   |--- Humidity3pm >  0.20
|   |   |   |   |   |   |   |   |   |   |--- Evaporation <= 0.02
|   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |   |   |--- Evaporation >  0.02
|   |   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |--- Humidity3pm >  0.28
|   |   |   |   |   |   |   |--- Sunshine <= 0.05
|   |   |   |   |   |   |   |   |--- WindGustSpeed <= 0.25
|   |   |   |   |   |   |   |   |   |--- Evaporation <= 0.01
|   |   |   |   |   |   |   |   |   |   |--- WindGustSpeed <= 0.23
|   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |   |   |--- WindGustSpeed >  0.23
|   |   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |   |   |--- Evaporation >  0.01
|   |   |   |   |   |   |   |   |   |   |--- Evaporation <= 0.07
|   |   |   |   |   |   |   |   |   |   |   |--- Temp3pm <= 0.34
|   |   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |   |   |   |--- Temp3pm >  0.34
|   |   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 11
```

```
|   |   |   |   |   |   |   |   |   |--- Evaporation >  0.07
|   |   |   |   |   |   |   |   |   |   |--- WindSpeed9am <= 0.12
|   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |   |   |--- WindSpeed9am >  0.12
|   |   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |--- WindGustSpeed >  0.25
|   |   |   |   |   |   |   |   |--- Pressure9am <= 0.56
|   |   |   |   |   |   |   |   |   |--- MinTemp <= 0.40
|   |   |   |   |   |   |   |   |   |   |--- WindDir9am_WNW <= 0.50
|   |   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |   |   |--- WindDir9am_WNW >  0.50
|   |   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |   |   |--- MinTemp >  0.40
|   |   |   |   |   |   |   |   |   |   |--- Humidity3pm <= 0.66
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 7
|   |   |   |   |   |   |   |   |   |   |--- Humidity3pm >  0.66
|   |   |   |   |   |   |   |   |   |   |   |--- truncated branch of depth 4
|   |   |   |   |   |   |   |   |   |--- Pressure9am >  0.56
|   |   |   |   |   |
```

> **EXERCISE**: Based on the above discussion, can you explain why the training accuracy is 100% whereas the validation accuracy is lower?

Feature Importance

Based on the gini index computations, a decision tree assigns an "importance" value to each feature. These values can be used to interpret the results given by a decision tree.

```
model.feature_importances_
```

```
array([3.48942086e-02, 3.23605486e-02, 5.91385668e-02, 2.49619907e-02,
       4.94652143e-02, 5.63334673e-02, 2.80205998e-02, 2.98128801e-02,
       4.02182908e-02, 2.61441297e-01, 3.44145027e-02, 6.20573699e-02,
       1.36406176e-02, 1.69229866e-02, 3.50001550e-02, 3.04064076e-02,
       2.24086587e-03, 2.08018104e-03, 1.27475954e-03, 7.26936324e-04,
       1.39779517e-03, 1.15264873e-03, 6.92808159e-04, 1.80675598e-03,
       1.08370901e-03, 1.19773895e-03, 8.87119103e-04, 2.15764220e-03,
       1.67094731e-03, 7.98919493e-05, 1.10558668e-03, 1.42008656e-03,
       4.10087635e-04, 1.09028115e-03, 1.44164766e-03, 9.08284767e-04,
       1.05770304e-03, 6.18133455e-04, 1.80387272e-03, 2.10403527e-03,
       2.74413333e-04, 7.31599405e-04, 1.35408990e-03, 1.54759332e-03,
       1.30917564e-03, 1.07134670e-03, 8.36408023e-04, 1.62662229e-03,
       1.00326116e-03, 2.16053455e-03, 8.46802258e-04, 1.88919081e-03,
```

```
9.29325203e-04, 1.29545157e-03, 1.27604831e-03, 5.12736888e-04,
1.38458902e-03, 3.97103931e-04, 1.03734689e-03, 1.44437047e-03,
1.75870184e-03, 1.42487857e-03, 2.78109569e-03, 2.00782698e-03,
2.80617652e-04, 1.61509734e-03, 1.64361598e-03, 2.36124112e-03,
3.05457932e-03, 2.33239534e-03, 2.78643875e-03, 2.16695261e-03,
3.41491352e-03, 2.30573542e-03, 2.28270604e-03, 2.34408118e-03,
2.26557332e-03, 2.54592702e-03, 2.75264499e-03, 2.83905192e-03,
2.49480561e-03, 1.54840338e-03, 2.50305095e-03, 2.53945388e-03,
2.28130055e-03, 3.80572180e-03, 2.58535069e-03, 3.10172224e-03,
2.54236791e-03, 2.50297796e-03, 2.06400988e-03, 2.52931192e-03,
2.07840517e-03, 1.77387278e-03, 1.78920555e-03, 2.77709687e-03,
2.42564566e-03, 2.26471887e-03, 1.73346117e-03, 2.23926957e-03,
2.47865244e-03, 2.31917387e-03, 3.21211861e-03, 2.92382975e-03,
2.24399274e-03, 3.68774754e-03, 3.87595982e-03, 3.20326068e-03,
2.53323550e-03, 2.40444844e-03, 2.26790411e-03, 2.19744009e-03,
2.28064147e-03, 2.88545323e-03, 2.05278867e-03, 1.12604304e-03,
2.86325849e-04, 1.32322128e-03, 1.72690480e-03])
```

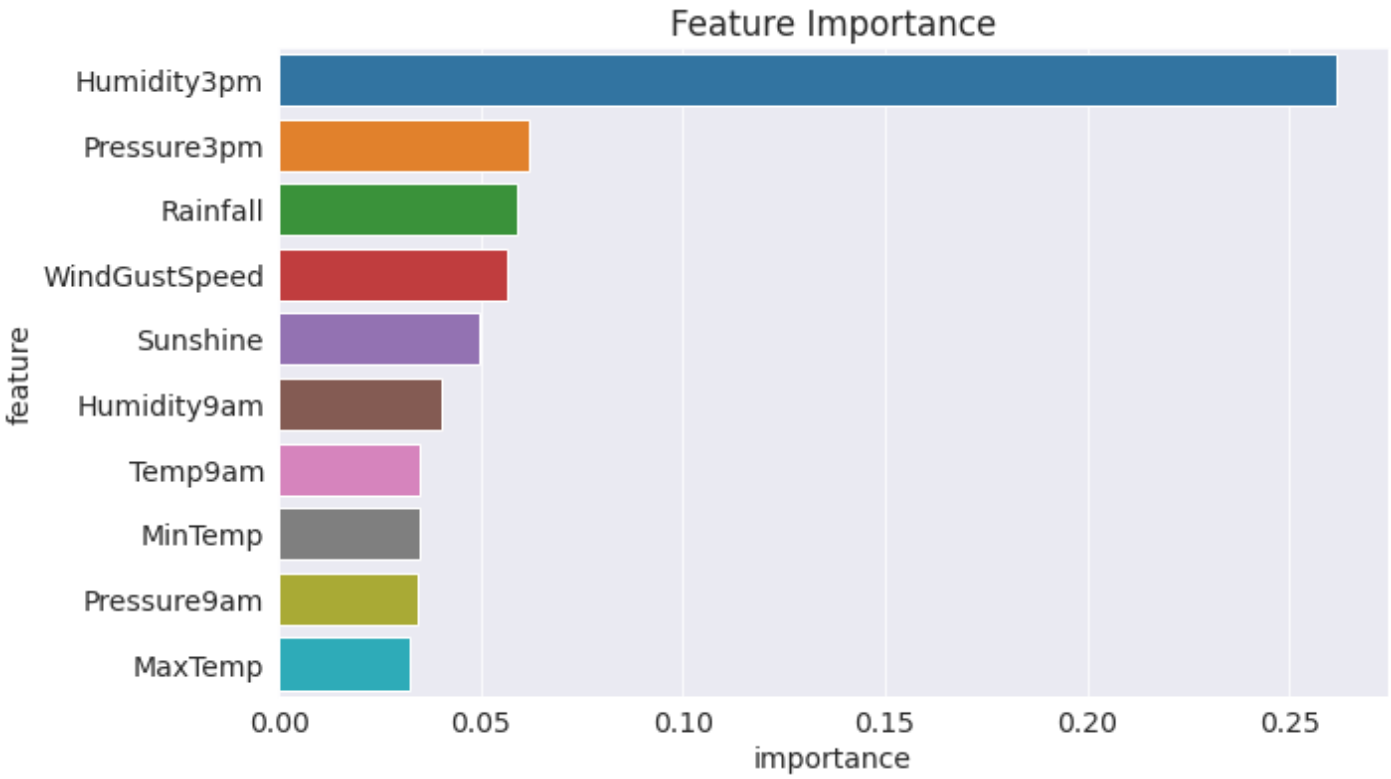
Let's turn this into a dataframe and visualize the most important features.

```
importance_df = pd.DataFrame({
    'feature': X_train.columns,
    'importance': model.feature_importances_
}).sort_values('importance', ascending=False)
```

```
importance_df.head(10)
```

	feature	importance
9	Humidity3pm	0.261441
11	Pressure3pm	0.062057
2	Rainfall	0.059139
5	WindGustSpeed	0.056333
4	Sunshine	0.049465
8	Humidity9am	0.040218
14	Temp9am	0.035000
0	MinTemp	0.034894
10	Pressure9am	0.034415
1	MaxTemp	0.032361

```
plt.title('Feature Importance')
sns.barplot(data=importance_df.head(10), x='importance', y='feature');
```



Let's save our work before continuing.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/aakashns/sklearn-decision-trees-random-forests
```

```
'https://jovian.ai/aakashns/sklearn-decision-trees-random-forests'
```

Hyperparameter Tuning and Overfitting

As we saw in the previous section, our decision tree classifier memorized all training examples, leading to a 100% training accuracy, while the validation accuracy was only marginally better than a dumb baseline model. This phenomenon is called overfitting, and in this section, we'll look at some strategies for reducing overfitting. The process of reducing overfitting is known as *regularlization*.

The `DecisionTreeClassifier` accepts several arguments, some of which can be modified to reduce overfitting.

```
?DecisionTreeClassifier
```

These arguments are called hyperparameters because they must be configured manually (as opposed to the parameters within the model which are *learned* from the data. We'll explore a couple of hyperparameters:

- `max_depth`
- `max_leaf_nodes`

max_depth

By reducing the maximum depth of the decision tree, we can prevent the tree from memorizing all training examples, which may lead to better generalization

```
model = DecisionTreeClassifier(max_depth=3, random_state=42)
```

```
model.fit(X_train, train_targets)
```

```
DecisionTreeClassifier(max_depth=3, random_state=42)
```

We can compute the accuracy of the model on the training and validation sets using `model.score`

```
model.score(X_train, train_targets)
```

```
0.8291308037337859
```

```
model.score(X_val, val_targets)
```

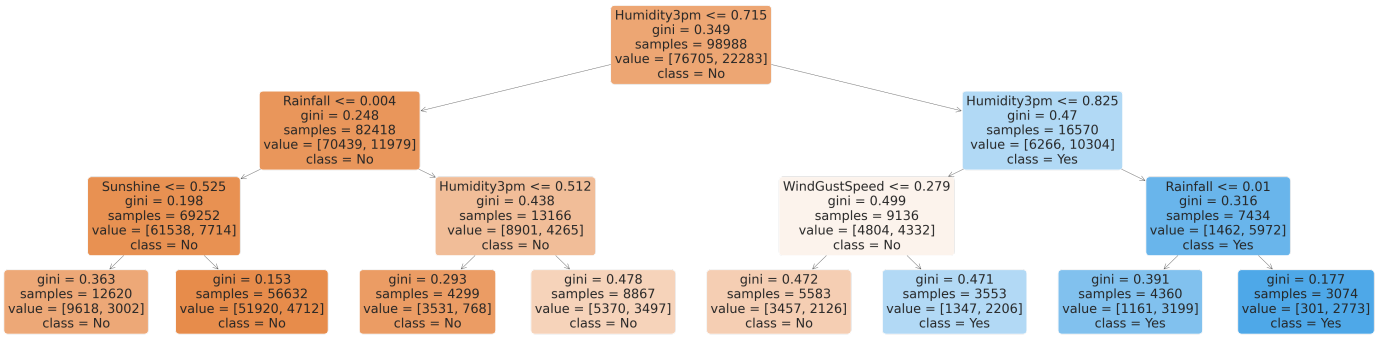
```
0.8334397307178921
```

Great, while the training accuracy of the model has gone down, the validation accuracy of the model has increased significantly.

```
model.classes_
```

```
array(['No', 'Yes'], dtype=object)
```

```
plt.figure(figsize=(80,20))
plot_tree(model, feature_names=X_train.columns, filled=True, rounded=True, class_names=
```



> **EXERCISE**: Study the decision tree diagram carefully and understand what each of the terms `gini`, `samples`, `value` and `class` mean.

```
print(export_text(model, feature_names=list(X_train.columns)))
```

```
|--- Humidity3pm <= 0.72
|   |--- Rainfall <= 0.00
```

```
|   |   |--- Sunshine <= 0.52
|   |   |   |--- class: No
|   |   |--- Sunshine >  0.52
|   |   |   |--- class: No
|   |--- Rainfall >  0.00
|   |   |--- Humidity3pm <= 0.51
|   |   |   |--- class: No
|   |   |--- Humidity3pm >  0.51
|   |   |   |--- class: No
|--- Humidity3pm >  0.72
|   |--- Humidity3pm <= 0.82
|   |   |--- WindGustSpeed <= 0.28
|   |   |   |--- class: No
|   |   |--- WindGustSpeed >  0.28
|   |   |   |--- class: Yes
|   |--- Humidity3pm >  0.82
|   |   |--- Rainfall <= 0.01
|   |   |   |--- class: Yes
|   |   |--- Rainfall >  0.01
|   |   |   |--- class: Yes
```

Let's experiment with different depths using a helper function.

```
def max_depth_error(md):
    model = DecisionTreeClassifier(max_depth=md, random_state=42)
    model.fit(X_train, train_targets)
    train_acc = 1 - model.score(X_train, train_targets)
    val_acc = 1 - model.score(X_val, val_targets)
    return {'Max Depth': md, 'Training Error': train_acc, 'Validation Error': val_acc}
```

```
%%time
errors_df = pd.DataFrame([max_depth_error(md) for md in range(1, 21)])
```

```
CPU times: user 40.4 s, sys: 59.6 ms, total: 40.5 s
Wall time: 40.4 s
```

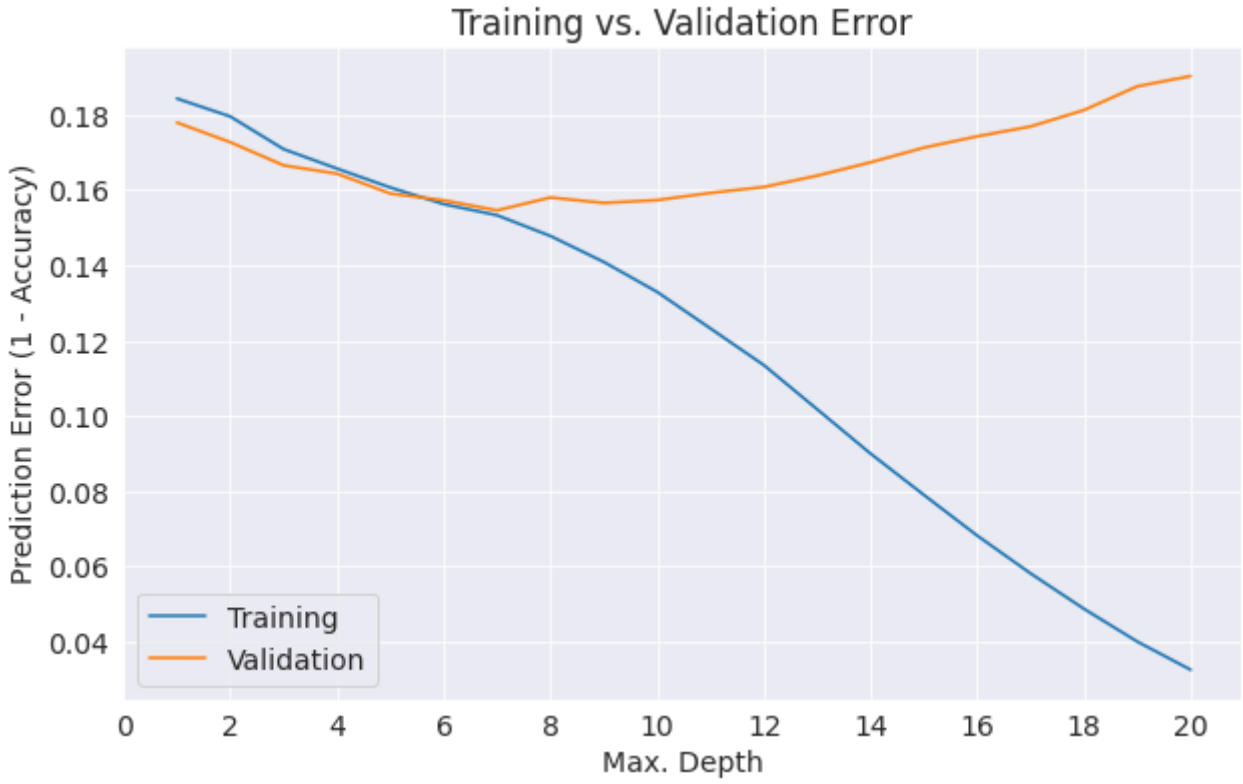
```
errors_df
```

	Max Depth	Training Error	Validation Error
0	1	0.184315	0.177935
1	2	0.179547	0.172712
2	3	0.170869	0.166560
3	4	0.165707	0.164355
4	5	0.160676	0.159074

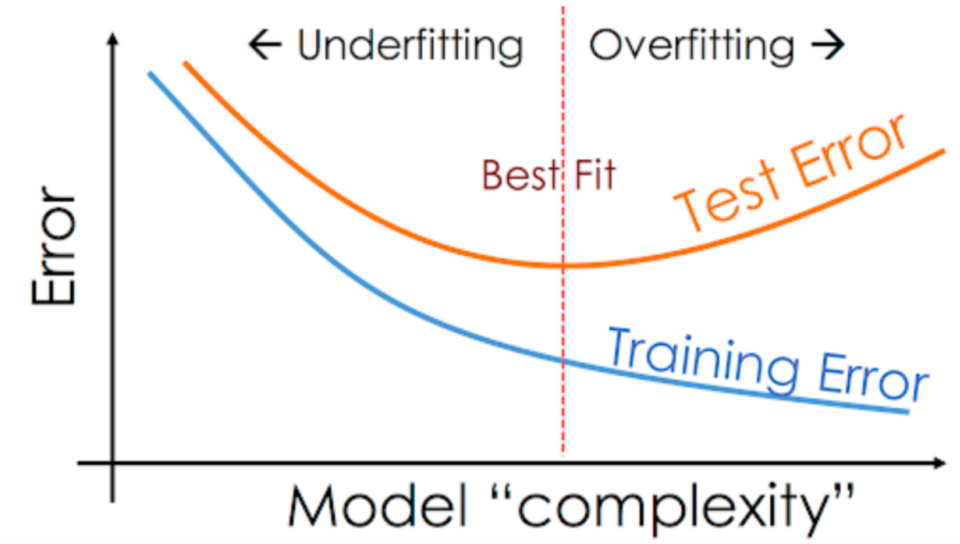
	Max Depth	Training Error	Validation Error
5	6	0.156271	0.157275
6	7	0.153312	0.154605
7	8	0.147806	0.158029
8	9	0.140906	0.156578
9	10	0.132945	0.157333
10	11	0.123227	0.159248
11	12	0.113489	0.160815
12	13	0.101750	0.163833
13	14	0.089981	0.167373
14	15	0.078999	0.171261
15	16	0.068180	0.174279
16	17	0.058138	0.176890
17	18	0.048733	0.181243
18	19	0.040025	0.187569
19	20	0.032539	0.190297

```
plt.figure()
plt.plot(errors_df['Max Depth'], errors_df['Training Error'])
plt.plot(errors_df['Max Depth'], errors_df['Validation Error'])
plt.title('Training vs. Validation Error')
plt.xticks(range(0,21, 2))
plt.xlabel('Max. Depth')
plt.ylabel('Prediction Error (1 - Accuracy)')
plt.legend(['Training', 'Validation'])
```

```
<matplotlib.legend.Legend at 0x7f9e3fd19a50>
```



This is a common pattern you'll see with all machine learning algorithms:



You'll often need to tune hyperparameters carefully to find the optimal fit. In the above case, it appears that a maximum depth of 7 results in the lowest validation error.

```
model = DecisionTreeClassifier(max_depth=7, random_state=42).fit(X_train, train_targets
model.score(X_val, val_targets)
```

```
0.8453949277465034
```

max_leaf_nodes

Another way to control the size of complexity of a decision tree is to limit the number of leaf nodes. This allows branches of the tree to have varying depths.

```
model = DecisionTreeClassifier(max_leaf_nodes=128, random_state=42)
```

```
model.fit(X_train, train_targets)
```

```
DecisionTreeClassifier(max_leaf_nodes=128, random_state=42)
```

```
model.score(X_train, train_targets)
```

```
0.8480421869317493
```

```
model.score(X_val, val_targets)
```

```
0.8442342290058615
```

```
model.tree_.max_depth
```

```
12
```

Notice that the model was able to achieve a greater depth of 12 for certain paths while keeping other paths shorter.

```
model_text = export_text(model, feature_names=list(X_train.columns))
print(model_text[:3000])
```

```
|--- Humidity3pm <= 0.72
|   |--- Rainfall <= 0.00
|   |   |--- Sunshine <= 0.52
|   |   |   |--- Pressure3pm <= 0.58
|   |   |   |   |--- WindGustSpeed <= 0.36
|   |   |   |   |   |--- Humidity3pm <= 0.28
|   |   |   |   |   |   |--- class: No
|   |   |   |   |   |--- Humidity3pm >  0.28
|   |   |   |   |   |   |--- Sunshine <= 0.05
|   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |--- Sunshine >  0.05
|   |   |   |   |   |   |   |--- Pressure3pm <= 0.43
|   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |--- Pressure3pm >  0.43
|   |   |   |   |   |   |   |   |--- Humidity3pm <= 0.57
|   |   |   |   |   |   |   |   |   |--- WindDir9am_NE <= 0.50
|   |   |   |   |   |   |   |   |   |   |--- WindDir9am_NNE <= 0.50
|   |   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |   |   |   |--- WindDir9am_NNE >  0.50
|   |   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |   |   |--- WindDir9am_NE >  0.50
|   |   |   |   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |   |   |--- Humidity3pm >  0.57
|   |   |   |   |   |   |   |   |   |--- MaxTemp <= 0.53
|   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |   |   |--- MaxTemp >  0.53
|   |   |   |   |   |   |   |   |   |   |--- Temp3pm <= 0.67
|   |   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |   |   |   |--- Temp3pm >  0.67
|   |   |   |   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |--- WindGustSpeed >  0.36
|   |   |   |   |   |--- Humidity3pm <= 0.45
|   |   |   |   |   |   |--- Sunshine <= 0.39
|   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |--- Sunshine >  0.39
|   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |--- Humidity3pm >  0.45
|   |   |   |   |   |   |--- Pressure3pm <= 0.49
|   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |--- Pressure3pm >  0.49
```

```
|   |   |   |   |   |   |   |--- class: Yes
|   |   |   |--- Pressure3pm >  0.58
|   |   |   |   |--- Pressure3pm <= 0.70
|   |   |   |   |   |--- Sunshine <= 0.32
|   |   |   |   |   |   |--- WindDir9am_N <= 0.50
|   |   |   |   |   |   |   |--- Humidity3pm <= 0.67
|   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |   |--- Humidity3pm >  0.67
|   |   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |--- WindDir9am_N >  0.50
|   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |--- Sunshine >  0.32
|   |   |   |   |   |   |--- WindGustSpeed <= 0.33
|   |   |   |   |   |   |   |--- class: No
|   |   |   |   |   |   |--- WindGustSpeed >  0.33
|   |   |   |   |   |   |   |--- class: No
|   |   |   |   |--- Pressure3pm >  0.70
|   |   |   |   |   |--- Location_CoffsHarbour <= 0.50
|   |   |   |   |   |   |--- class: No
|   |   |   |   |   |--- Location_CoffsHarbour >  0.50
|   |   |   |   |   |   |--- class: No
|   |   |   |--- Sunshine >  0.52
|   |   |   |
```

> **EXERCISE**: Find the combination of `max_depth` and `max_leaf_nodes` that results in the highest validation accuracy.

> **EXERCISE**: Explore and experiment with other arguments of `DecisionTree` . Refer to the docs for details: https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html

> **EXERCISE**: A more advanced technique (but less commonly used technique) for reducing overfitting in decision trees is known as cost-complexity pruning. Learn more about it here: https://scikit-learn.org/stable/auto_examples/tree/plot_cost_complexity_pruning.html . Implement cost complexity pruning. Do you see any improvement in the validation accuracy?

Let's save our work before continuing.

```
jovian.commit()
```

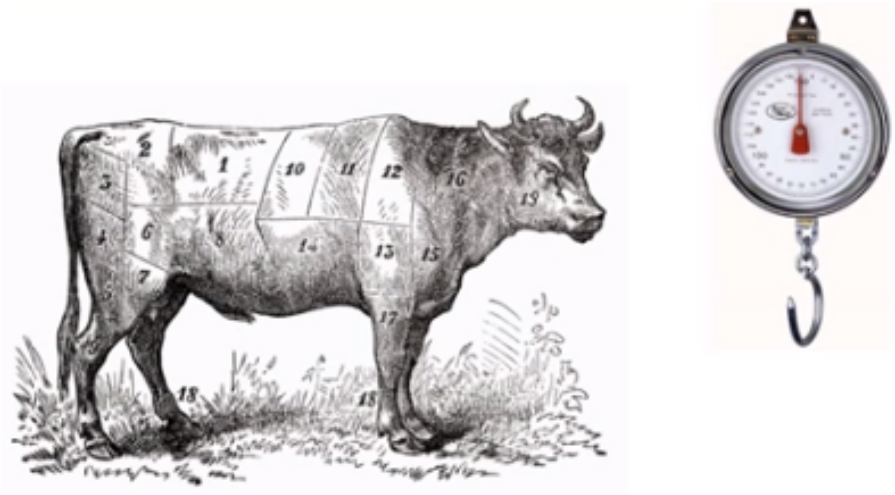
```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/aakashns/sklearn-decision-trees-random-forests
```

```
'https://jovian.ai/aakashns/sklearn-decision-trees-random-forests'
```

Training a Random Forest

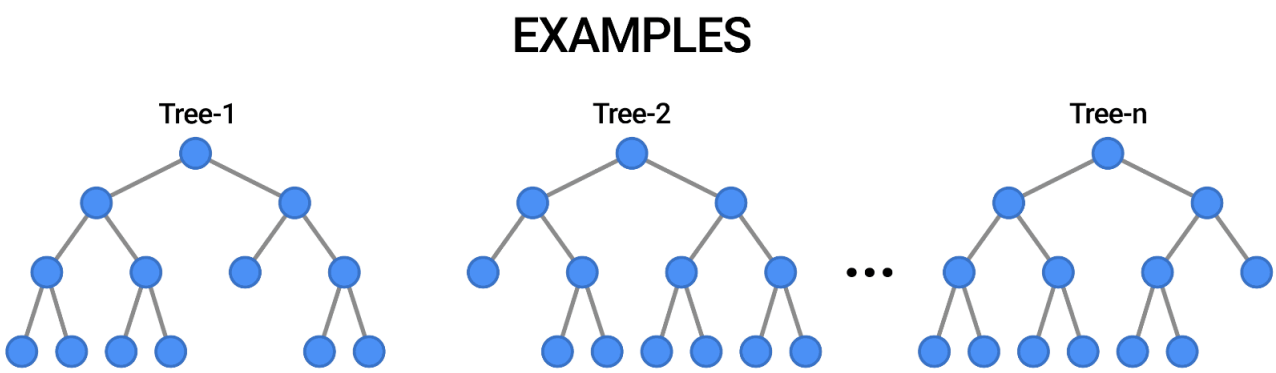
While tuning the hyperparameters of a single decision tree may lead to some improvements, a much more effective strategy is to combine the results of several decision trees trained with slightly different parameters. This is called a random forest model.

The key idea here is that each decision tree in the forest will make different kinds of errors, and upon averaging, many of their errors will cancel out. This idea is also commonly known as the "wisdom of the crowd":



average of 800 guesses = 1,197
actual weight of the ox = 1,198

A random forest works by averaging/combining the results of several decision trees:



We'll use the `RandomForestClassifier` class from `sklearn.ensemble`.

```
from sklearn.ensemble import RandomForestClassifier
```

```
model = RandomForestClassifier(n_jobs=-1, random_state=42)
```

`n_jobs` allows the random forest to use mutiple parallel workers to train decision trees, and `random_state=42` ensures that the we get the same results for each execution.

```
%%time
model.fit(X_train, train_targets)
```

```
CPU times: user 37.6 s, sys: 224 ms, total: 37.8 s
Wall time: 19.3 s
```

```
RandomForestClassifier(n_jobs=-1, random_state=42)
```

```
model.score(X_train, train_targets)
```

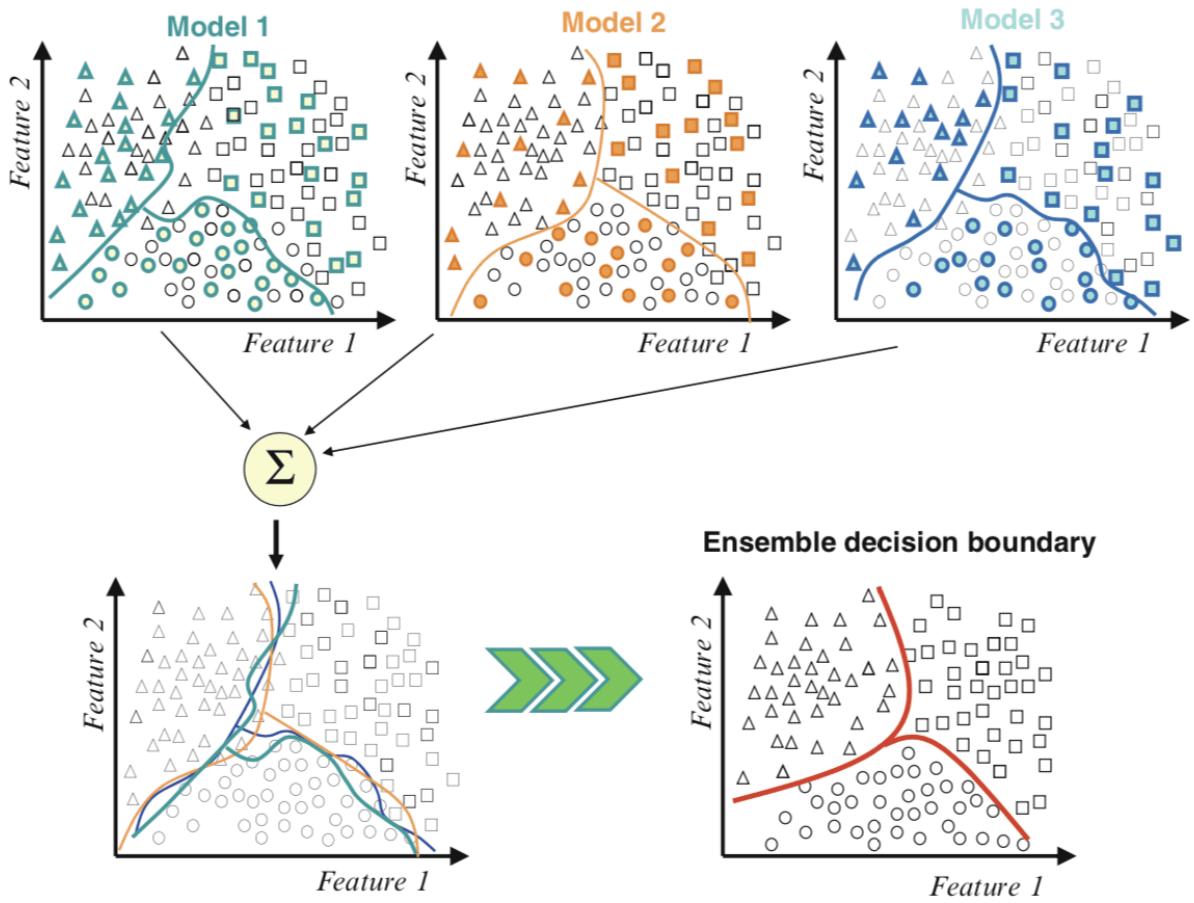
```
0.9999494888269285
```

```
model.score(X_val, val_targets)
```

```
0.8566537055307295
```

Once again, the training accuracy is almost 100%, but this time the validation accuracy is much better. In fact, it is better than the best single decision tree we had trained so far. Do you see the power of random forests?

This general technique of combining the results of many models is called "ensembling", it works because most errors of individual models cancel out on averaging. Here's what it looks like visually:



We can also look at the probabilities for the predictions. The probability of a class is simply the fraction of trees which that predicted the given class.

```
train_probs = model.predict_proba(X_train)
train_probs
```

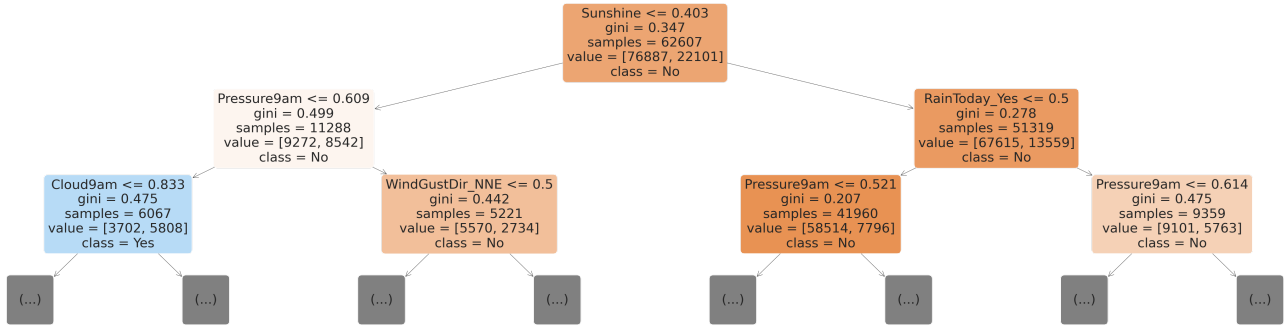
```
array([[0.93, 0.07],
       [1.  , 0.  ],
       [0.99, 0.01],
       ...,
       [0.99, 0.01],
       [1.  , 0.  ],
       [0.96, 0.04]])
```

We can can access individual decision trees using `model.estimators_`

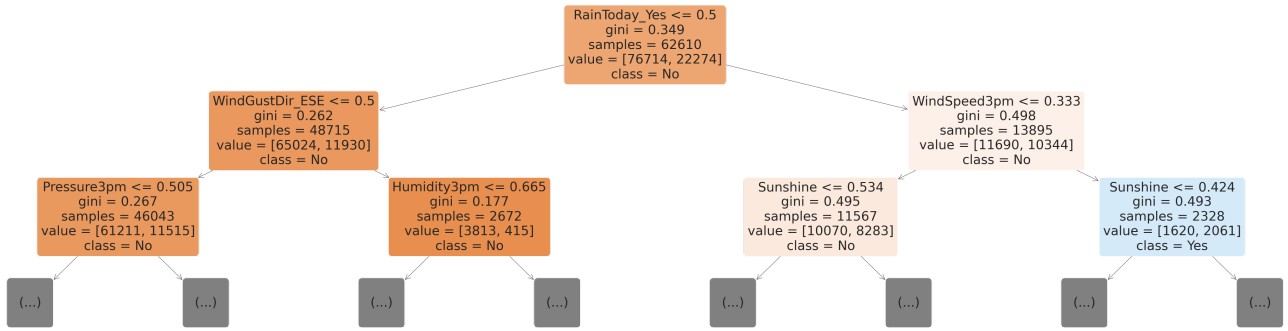
```
model.estimators_[0]
```

```
DecisionTreeClassifier(max_features='auto', random_state=1608637542)
```

```
plt.figure(figsize=(80,20))
plot_tree(model.estimators_[0], max_depth=2, feature_names=X_train.columns, filled=True
```




```
plt.figure(figsize=(80,20))
plot_tree(model.estimators_[20], max_depth=2, feature_names=X_train.columns, filled=Tru
```




```
len(model.estimators_)
```

```
100
```

> **EXERCISE**: Verify that none of the individual decision trees have a better validation accuracy than the random forest.

Just like decision tree, random forests also assign an "importance" to each feature, by combining the importance values from individual trees.

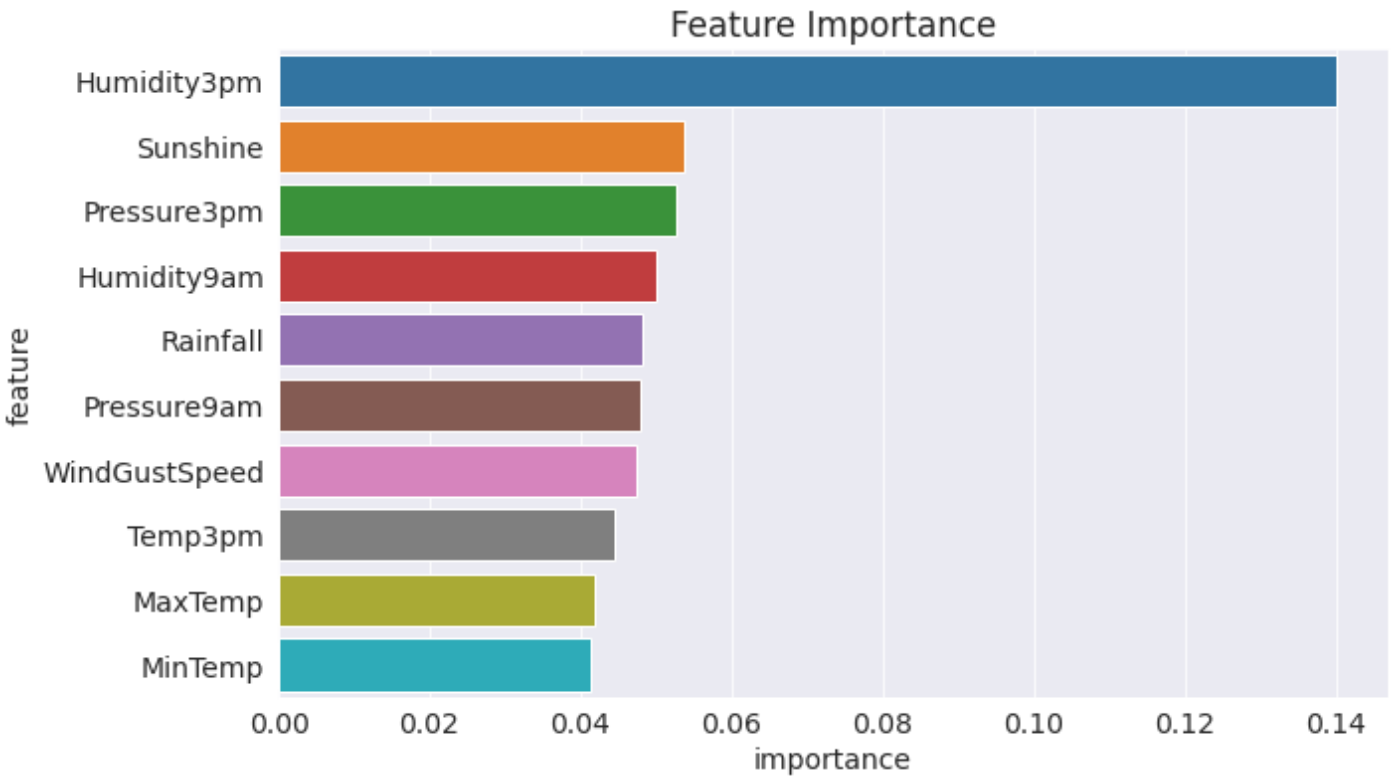
```
importance_df = pd.DataFrame({
    'feature': X_train.columns,
    'importance': model.feature_importances_
}).sort_values('importance', ascending=False)
```

```
importance_df.head(10)
```

	feature	importance
9	Humidity3pm	0.139904
4	Sunshine	0.053696

	feature	importance
11	Pressure3pm	0.052713
8	Humidity9am	0.050051
2	Rainfall	0.048077
10	Pressure9am	0.047944
5	WindGustSpeed	0.047477
15	Temp3pm	0.044379
1	MaxTemp	0.041865
0	MinTemp	0.041199

```
plt.title('Feature Importance')
sns.barplot(data=importance_df.head(10), x='importance', y='feature');
```



Notice that the distribution is a lot less skewed than that for a single decision tree.

Let's save our work before continuing.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/aakashns/sklearn-decision-trees-random-forests
```

```
'https://jovian.ai/aakashns/sklearn-decision-trees-random-forests'
```

Hyperparameter Tuning with Random Forests

Just like decision trees, random forests also have several hyperparameters. In fact many of these hyperparameters are applied to the underlying decision trees.

Let's study some the hyperparameters for random forests. You can learn more about them here: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

```
?RandomForestClassifier
```

Let's create a base model with which we can compare models with tuned hyperparameters.

```
base_model = RandomForestClassifier(random_state=42, n_jobs=-1).fit(X_train, train_targ
```

```
base_train_acc = base_model.score(X_train, train_targets)
base_val_acc = base_model.score(X_val, val_targets)
```

```
base_accs = base_train_acc, base_val_acc
base_accs
```

```
(0.9999494888269285, 0.8566537055307295)
```

We can use this as a benchmark for hyperparmeter tuning.

n_estimators

This argument controls the number of decision trees in the random forest. The default value is 100. For larger datasets, it helps to have a greater number of estimators. As a general rule, try to have as few estimators as needed.

10 estimators

```
model = RandomForestClassifier(random_state=42, n_jobs=-1, n_estimators=10)
```

```
model.fit(X_train, train_targets)
```

```
RandomForestClassifier(n_estimators=10, n_jobs=-1, random_state=42)
```

```
model.score(X_train, train_targets), model.score(X_val, val_targets)
```

```
(0.986958015112943, 0.8485868492832686)
```

```
base_accs
```

```
(0.9999494888269285, 0.8566537055307295)
```

500 estimators

```
model = RandomForestClassifier(random_state=42, n_jobs=-1, n_estimators=500)
model.fit(X_train, train_targets)
```

```
RandomForestClassifier(n_estimators=500, n_jobs=-1, random_state=42)
```

```
model.score(X_train, train_targets)
```

```
0.9999797955307714
```

```
model.score(X_val, val_targets)
```

```
0.8577563693343393
```

```
base_accs
```

```
(0.9999494888269285, 0.8566537055307295)
```

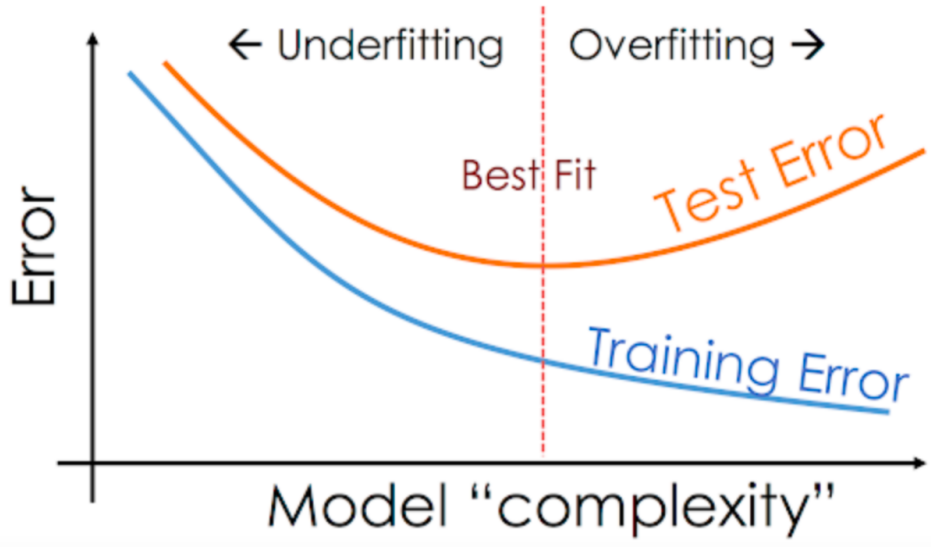
> **EXERCISE**: Vary the value of `n_estimators` and plot the graph between training error and validation error. What is the optimal value of `n_estimators` ?

```

```

max_depth and max_leaf_nodes

These arguments are passed directly to each decision tree, and control the maximum depth and max. no leaf nodes of each tree respectively. By default, no maximum depth is specified, which is why each tree has a training accuracy of 100%. You can specify a `max_depth` to reduce overfitting.



Let's define a helper function `test_params` to make it easy to test hyperparameters.

```
def test_params(**params):
    model = RandomForestClassifier(random_state=42, n_jobs=-1, **params).fit(X_train, t
    return model.score(X_train, train_targets), model.score(X_val, val_targets)
```

Let's test a few values of `max_depth` and `max_leaf_nodes` .

```
test_params(max_depth=5)
```

```
(0.8197862367155615, 0.8240961058557251)
```

```
test_params(max_depth=26)
```

```
(0.9814826039519942, 0.8572340549010504)
```

```
test_params(max_leaf_nodes=2**5)
```

```
(0.8314341132258456, 0.833904010214149)
```

```
test_params(max_leaf_nodes=2**20)
```

```
(0.9999494888269285, 0.8556671116011839)
```

```
base_accs # no max depth or max leaf nodes
```

```
(0.9999494888269285, 0.8566537055307295)
```

The optimal values of `max_depth` and `max_leaf_nodes` lies somewhere between 0 and unbounded.

> **EXERCISE**: Vary the value of `max_depth` and plot the graph between training error and validation error. What is the optimal value of `max_depth` ? Do the same for `max_leaf_nodes` .

```

```

```

```

max_features

Instead of picking all features (columns) for every split, we can specify that only a fraction of features be chosen randomly to figure out a split.

max_features : *{"auto", "sqrt", "log2"}, int or float, default="auto"*
The number of features to consider when looking for the best split:

- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and `round(max_features * n_features)` features are considered at each split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)` (same as "auto").
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.

Note: the search for a split does not stop until at least one valid partition of the node samples is found, even if it requires to effectively inspect more than `max_features` features.

Notice that the default value `auto` causes only $\sqrt{n}$ out of total features (n) to be chosen randomly at each split. This is the reason each decision tree in the forest is different. While it may seem counterintuitive, choosing all features for every split of every tree will lead to identical trees, so the random forest will not generalize well.

```
test_params(max_features='log2')
```

```
(0.9999595910615429, 0.8558992513493123)
```

```
test_params(max_features=3)
```

```
(0.9999494888269285, 0.8543323080494458)
```

```
test_params(max_features=6)
```

```
(0.9999595910615429, 0.8558992513493123)
```

```
base_accs
```

```
(0.9999494888269285, 0.8566537055307295)
```

> **EXERCISE**: Find the optimal values of `max_features` for this dataset.

```

```

`min_samples_split` and `min_samples_leaf`

By default, the decision tree classifier tries to split every node that has 2 or more. You can increase the values of these arguments to change this behavior and reduce overfitting, especially for very large datasets.

```
test_params(min_samples_split=3, min_samples_leaf=2)
```

```
(0.9625005051117307, 0.8565956705936975)
```

```
test_params(min_samples_split=100, min_samples_leaf=60)
```

```
(0.8495676243585081, 0.8451047530613429)
```

```
base_accs
```

```
(0.9999494888269285, 0.8566537055307295)
```

> **EXERCISE**: Find the optimal values of `min_samples_split and min_samples_leaf .`

```

```

```
```

min_impurity_decrease

This argument is used to control the threshold for splitting nodes. A node will be split if this split induces a decrease of the impurity (Gini index) greater than or equal to this value. It's default value is 0, and you can increase it to reduce overfitting.

```
test_params(min_impurity_decrease=1e-7)
```

```
(0.9996060128500425, 0.8561313910974406)
```

```
test_params(min_impurity_decrease=1e-2)
```

```
(0.774891906089627, 0.7882885497069235)
```

```
base_accs
```

```
(0.9999494888269285, 0.8566537055307295)
```

> **EXERCISE**: Find the optimal values of `min_impurity_decrease` for this dataset.

```
```

bootstrap, max_samples

By default, a random forest doesn't use the entire dataset for training each decision tree. Instead it applies a technique called bootstrapping. For each tree, rows from the dataset are picked one by one randomly, with replacement i.e. some rows may not show up at all, while some rows may show up multiple times.

> **bootstrap : *bool, default=True***
> Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

Bootstrapping helps the random forest generalize better, because each decision tree only sees a fraction of th training set, and some rows randomly get higher weightage than others.

```
test_params(bootstrap=False)
```

```
(0.9999797955307714, 0.8567697754047937)
```

```
base_accs
```

```
(0.9999494888269285, 0.8566537055307295)
```

When bootstrapping is enabled, you can also control the number or fraction of rows to be considered for each bootstrap using `max_samples`. This can further generalize the model.

max_samples : *int or float, default=None*

If bootstrap is True, the number of samples to draw from X to train each base estimator.

- If None (default), then draw `X.shape[0]` samples.
- If int, then draw `max_samples` samples.
- If float, then draw `max_samples * X.shape[0]` samples. Thus, `max_samples` should be in the interval `(0, 1)`.

```
test_params(max_samples=0.9)
```

```
(0.9997676486038711, 0.8565376356566653)
```

```
base_accs
```

```
(0.9999494888269285, 0.8566537055307295)
```

Learn more about bootstrapping here: https://towardsdatascience.com/what-is-out-of-bag-oob-score-in-random-forest-a7fa23d710

class_weight

```
model.classes_
```

```
array(['No', 'Yes'], dtype=object)
```

```
test_params(class_weight='balanced')
```

```
(0.9999494888269285, 0.8543903429864779)
```

```
test_params(class_weight={'No': 1, 'Yes': 2})
```

```
(0.9999595910615429, 0.8558412164122802)
```

```
base_accs
```

```
(0.9999494888269285, 0.8566537055307295)
```

> **EXERCISE**: Find the optimal value of `class_weight` for this dataset.

```

```

Putting it together

Let's train a random forest with customized hyperparameters based on our learnings. Of course, different hyperpraams

```
model = RandomForestClassifier(n_jobs=-1,
                               random_state=42,
```

```
                                n_estimators=500,
                                max_features=7,
                                max_depth=30,
                                class_weight={'No': 1, 'Yes': 1.5})
```

```
model.fit(X_train, train_targets)
```

```
RandomForestClassifier(class_weight={'No': 1, 'Yes': 1.5}, max_depth=30,
                       max_features=7, n_estimators=500, n_jobs=-1,
                       random_state=42)
```

```
model.score(X_train, train_targets), model.score(X_val, val_targets)
```

```
(0.9920192346547057, 0.8563054959085369)
```

```
base_accs
```

```
(0.9999494888269285, 0.8566537055307295)
```

We've increased the accuracy from 84.5% with a single decision tree to 85.7% with a well-tuned random forest. Depending on the dataset and the kind of problem, you may or may not a see a significant improvement with hyperparameter tuning.

This could be due to any of the following reasons:

- We may not have found the right mix of hyperparameters to regularize (reduce overfitting) the model properly, and we should keep trying to improve the model.
- We may have reached the limits of the modeling technique we're currently using (Random Forests), and we should try another modeling technique e.g. gradient boosting.
- We may have reached the limits of what we can predict using the given amount of data, and we may need more data to improve the model.
- We may have reached the limits of how well we can predict whether it will rain tomorrow using the given weather measurements, and we may need more features (columns) to further improve the model. In many cases, we can also generate new features using existing features (this is called feature engineering).
- Whether it will rain tomorrow may be an inherently random or chaotic phenomenon which simply cannot be predicted beyond a certain accuracy any amount of data for any number of weather measurements with any modeling technique.

Remember that ultimately all models are wrong, but some are useful. If you can rely on the model we've created today to make a travel decision for tomorrow, then the model is useful, even though it may sometimes be wrong.

```
raw_df
```

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindD
0	2008-12-01	Albury	13.4	22.9	0.6	NaN	NaN	W	44.0	
1	2008-12-02	Albury	7.4	25.1	0.0	NaN	NaN	WNW	44.0	

	Date	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSpeed	WindD
2	2008-12-03	Albury	12.9	25.7	0.0	NaN	NaN	WSW	46.0	
3	2008-12-04	Albury	9.2	28.0	0.0	NaN	NaN	NE	24.0	
4	2008-12-05	Albury	17.5	32.3	1.0	NaN	NaN	W	41.0	
...	...	...	...	...	...	...	...	...	...	
145454	2017-06-20	Uluru	3.5	21.8	0.0	NaN	NaN	E	31.0	
145455	2017-06-21	Uluru	2.8	23.4	0.0	NaN	NaN	E	31.0	
145456	2017-06-22	Uluru	3.6	25.3	0.0	NaN	NaN	NNW	22.0	
145457	2017-06-23	Uluru	5.4	26.9	0.0	NaN	NaN	N	37.0	
145458	2017-06-24	Uluru	7.8	27.0	0.0	NaN	NaN	SE	28.0	

142193 rows × 23 columns

> **EXERCISE**: Experiment with the hyperparameters of the random forest classifier, and try to maximize the validation accuracy.

```
```

```
```

Finally, let's also compute the accuracy of our model on the test set.

```
model.score(X_test, test_targets)
```

```
0.8451913451913452
```

Notice that the test accuracy is lower

Let's save our work before continuing.

```
jovian.commit()
```

```
[jovian] Detected Colab notebook...
[jovian] Uploading colab notebook to Jovian...
Committed successfully! https://jovian.ai/aakashns/sklearn-decision-trees-random-forests
```

```
'https://jovian.ai/aakashns/sklearn-decision-trees-random-forests'
```

Making Predictions on New Inputs

Let's define a helper function to make predictions on new inputs.

```
def predict_input(model, single_input):
    input_df = pd.DataFrame([single_input])
    input_df[numeric_cols] = imputer.transform(input_df[numeric_cols])
    input_df[numeric_cols] = scaler.transform(input_df[numeric_cols])
    input_df[encoded_cols] = encoder.transform(input_df[categorical_cols])
    X_input = input_df[numeric_cols + encoded_cols]
    pred = model.predict(X_input)[0]
    prob = model.predict_proba(X_input)[0][list(model.classes_).index(pred)]
    return pred, prob
```

```
new_input = {'Date': '2021-06-19',
             'Location': 'Launceston',
             'MinTemp': 23.2,
             'MaxTemp': 33.2,
             'Rainfall': 10.2,
             'Evaporation': 4.2,
             'Sunshine': np.nan,
             'WindGustDir': 'NNW',
             'WindGustSpeed': 52.0,
             'WindDir9am': 'NW',
             'WindDir3pm': 'NNE',
             'WindSpeed9am': 13.0,
             'WindSpeed3pm': 20.0,
             'Humidity9am': 89.0,
             'Humidity3pm': 58.0,
             'Pressure9am': 1004.8,
             'Pressure3pm': 1001.5,
             'Cloud9am': 8.0,
             'Cloud3pm': 5.0,
             'Temp9am': 25.7,
             'Temp3pm': 33.0,
             'RainToday': 'Yes'}
```

```
predict_input(model, new_input)
```

```
('Yes', 0.7608595348304203)
```

> **EXERCISE**: Try changing the values in `new_input` and observe how the predictions and probabilities change. Try different values of location, temperature, humidity, pressure etc. Try to get an *intuitive feel* of which columns have the greatest effect on the result of the model.

```
raw_df.Location.unique()
```

```
array(['Albury', 'BadgerysCreek', 'Cobar', 'CoffsHarbour', 'Moree',
       'Newcastle', 'NorahHead', 'NorfolkIsland', 'Penrith', 'Richmond',
       'Sydney', 'SydneyAirport', 'WaggaWagga', 'Williamtown',
       'Wollongong', 'Canberra', 'Tuggeranong', 'MountGinini', 'Ballarat',
       'Bendigo', 'Sale', 'MelbourneAirport', 'Melbourne', 'Mildura',
       'Nhil', 'Portland', 'Watsonia', 'Dartmoor', 'Brisbane', 'Cairns',
       'GoldCoast', 'Townsville', 'Adelaide', 'MountGambier', 'Nuriootpa',
       'Woomera', 'Albany', 'Witchcliffe', 'PearceRAAF', 'PerthAirport',
       'Perth', 'SalmonGums', 'Walpole', 'Hobart', 'Launceston',
       'AliceSprings', 'Darwin', 'Katherine', 'Uluru'], dtype=object)
```

Saving and Loading Trained Models

We can save the parameters (weights and biases) of our trained model to disk, so that we needn't retrain the model from scratch each time we wish to use it. Along with the model, it's also important to save imputers, scalers, encoders and even column names. Anything that will be required while generating predictions using the model should be saved.

We can use the `joblib` module to save and load Python objects on the disk.

```
import joblib
```

```
aussie_rain = {
    'model': model,
    'imputer': imputer,
    'scaler': scaler,
    'encoder': encoder,
    'input_cols': input_cols,
    'target_col': target_col,
    'numeric_cols': numeric_cols,
    'categorical_cols': categorical_cols,
    'encoded_cols': encoded_cols
}
```

```
joblib.dump(aussie_rain, 'aussie_rain.joblib')
```

```
['aussie_rain.joblib']
```

The object can be loaded back using `joblib.load`

```
aussie_rain2 = joblib.load('aussie_rain.joblib')
```

```
test_preds2 = aussie_rain2['model'].predict(X_test)
accuracy_score(test_targets, test_preds2)
```

Let's save our work before continuing.

```
jovian.commit()
```

Summary and References

The following topics were covered in this tutorial:

- Downloading a real-world dataset
- Preparing a dataset for training
- Training and interpreting decision trees
- Training and interpreting random forests
- Overfitting, hyperparameter tuning & regularization
- Making predictions on single inputs

We also introduced the following terms:

- Decision tree
- Random forest
- Overfitting
- Hyperparameter
- Hyperparameter tuning
- Regularization
- Ensembling
- Generalization
- Bootstrapping

Check out the following resources to learn more:

- https://scikit-learn.org/stable/modules/tree.html
- https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
- https://www.kaggle.com/willkoehrsen/start-here-a-gentle-introduction
- https://www.kaggle.com/willkoehrsen/introduction-to-manual-feature-engineering
- https://www.kaggle.com/willkoehrsen/intro-to-model-tuning-grid-and-random-search
- https://www.kaggle.com/c/home-credit-default-risk/discussion/64821

Revision Questions

1. What is a decision tree model?
2. What is `DecisionTreeClassifier()`?
3. Can we use decision tree only for Classifier?
4. How can you visualize the decision tree?
5. What is `max_depth` in decision tree?
6. What is gini index?
7. What is feature importance?

8. What is overfitting? What could be the reason for overfitting?
9. What is hyperparameter tuning?
10. What is one way to control the complexity of the decision tree?
11. What is a random forest model?
12. What is `RandomForestClassifier()`?
13. What is `model.score()`?
14. What is generalization?
15. What is ensembling?
16. What is `n_estimators` in hyperparameter tuning of random forests?
17. What is underfitting?
18. What does `max_features` parameter do?
19. What are some features that help in controlling the threshold for splitting nodes in decision tree?
20. What is bootstrapping? What is `max_samples` parameter in bootstrapping?
21. What is `class_weight` parameter?
22. You may or may not a see a significant improvement in the accuracy score with hyperparameter tuning. What could be the possible reasons for that?