

# Decision Trees and Random Forests

## Problem Statement:

Predict next-day rain by training classification models on the target variable RainTomorrow.

We can read the data back preprocessed datasets for training, validation, and testing using `pd.read_parquet`:

- Splitting a dataset into training, validation & test sets
- Filling/imputing missing values in numeric columns
- Scaling numeric features to a (0,1) range
- Encoding categorical columns as one-hot vectors

```
In [1]: import pandas as pd
```

```
In [2]: train_inputs = pd.read_parquet('train_inputs.parquet')
val_inputs = pd.read_parquet('val_inputs.parquet')
test_inputs = pd.read_parquet('test_inputs.parquet')

train_targets = pd.read_parquet('train_targets.parquet')["RainTomorrow"]
val_targets = pd.read_parquet('val_targets.parquet')["RainTomorrow"]
test_targets = pd.read_parquet('test_targets.parquet')["RainTomorrow"]
```

```
In [3]: print('train_inputs:', train_inputs.shape)
print('train_targets:', train_targets.shape)
print('val_inputs:', val_inputs.shape)
print('val_targets:', val_targets.shape)
print('test_inputs:', test_inputs.shape)
print('test_targets:', test_targets.shape)
```

```
train_inputs: (9788, 123)
train_targets: (9788,)
val_inputs: (1700, 123)
val_targets: (1700,)
test_inputs: (2591, 123)
test_targets: (2591,)
```

```
In [4]: train_targets
```

```
Out[4]: 126989    No
40299     No
28288     No
134636    No
130983    Yes
...
50356     No
122446    Yes
49288     No
119280    No
118137    No
Name: RainTomorrow, Length: 9788, dtype: object
```

```
In [5]: test_targets
```

```
Out[5]: 80721      No
82017      Yes
```

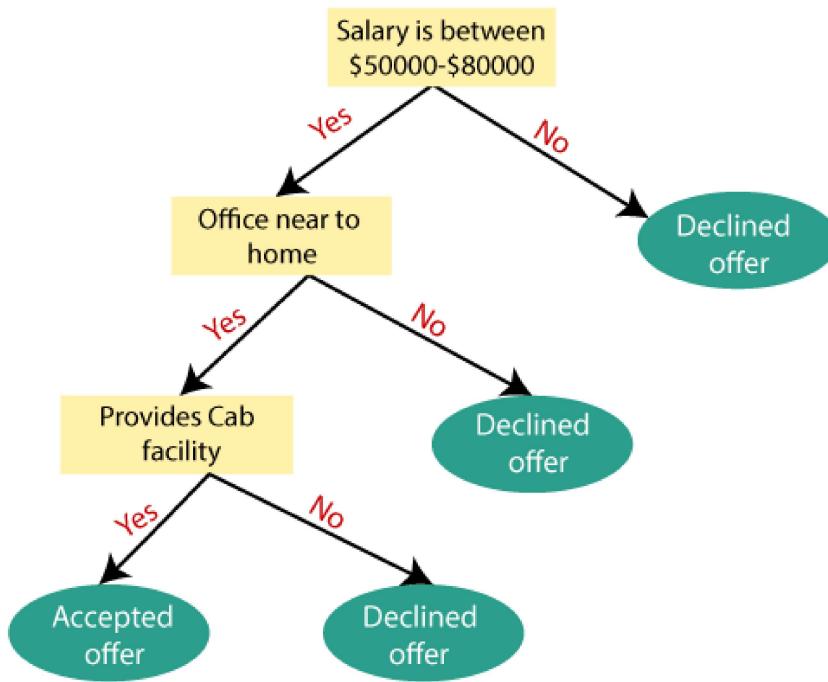
```

8594 / Yes
23750 No
89908 No
93275 No
...
143525 No
93147 No
120606 Yes
39511 No
48783 No
Name: RainTomorrow, Length: 2591, dtype: object

```

## 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 up with criteria manually.

In [6]: `train_inputs`

Out[6]:	Location	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustDir	WindGustSp
<b>126989</b>	Walpole	0.437824	0.397959	0.031655	0.094322	0.534317	NW	0.190
<b>40299</b>	Williamtown	0.694301	0.932653	0.000000	0.168966	0.895105	WNW	0.427
<b>28288</b>	Richmond	0.634715	0.618367	0.000000	0.094322	0.534317	NNE	0.136
<b>134636</b>	Launceston	0.287565	0.314286	0.021103	0.094322	0.534317	SSW	0.054
<b>130983</b>	Hobart	0.297927	0.467347	0.000000	0.048276	0.272727	NW	0.390
...	...	...	...	...	...	...	...	...
<b>50356</b>	Tuggeranong	0.147668	0.373469	0.000000	0.094322	0.534317	W	0.209
<b>122446</b>	Perth	0.461140	0.483673	0.000000	0.058621	0.223776	N	0.236
<b>49288</b>	Tuggeranong	0.386010	0.455102	0.002878	0.094322	0.534317	NNW	0.327

119280	PerthAirport	0.440415	0.538776	0.000000	0.082759	0.699301	NW	0.290
118137	PerthAirport	0.313472	0.448980	0.000000	0.041379	0.636364	SSW	0.236

9788 rows × 123 columns

```
In [7]: import numpy as np
```

```
In [8]: numeric_cols = train_inputs.select_dtypes(include=np.number).columns.tolist()  
numeric_cols
```

```
Out[8]: ['MinTemp',  
        'MaxTemp',  
        'Rainfall',  
        'Evaporation',  
        'Sunshine',  
        'WindGustSpeed',  
        'WindSpeed9am',  
        'WindSpeed3pm',  
        'Humidity9am',  
        'Humidity3pm',  
        'Pressure9am',  
        'Pressure3pm',  
        'Cloud9am',  
        'Cloud3pm',  
        'Temp9am',  
        'Temp3pm',  
        'Location_Adelaide',  
        'Location_Albany',  
        'Location_Albury',  
        'Location_AliceSprings',  
        'Location_BadgerysCreek',  
        'Location_Ballarat',  
        'Location_Bendigo',  
        'Location_Brisbane',  
        'Location_Cairns',  
        'Location_Canberra',  
        'Location_Cobar',  
        'Location_CoffsHarbour',  
        'Location_Dartmoor',  
        'Location_Darwin',  
        'Location_GoldCoast',  
        'Location_Hobart',  
        'Location_Katherine',  
        'Location_Launceston',  
        'Location_Melbourne',  
        'Location_MelbourneAirport',  
        'Location_Mildura',  
        'Location_Moree',  
        'Location_MountGambier',  
        'Location_MountGinini',  
        'Location_Newcastle',  
        'Location_Nhil',  
        'Location_NorahHead',  
        'Location_NorfolkIsland',  
        'Location_Nuriootpa',  
        'Location_PearceRAAF',  
        'Location_Penrith',  
        'Location_Perth',  
        'Location_PerthAirport',  
        'Location_Portland',  
        'Location_Richmond',
```

```
'Location_Sale',
'Location_SalmonGums',
'Location_Sydney',
'Location_SydneyAirport',
'Location_Townsville',
'Location_Tuggeranong',
'Location_Uluru',
'Location_WaggaWagga',
'Location_Walpole',
'Location_Watsonia',
'Location_Williamtown',
'Location_Witchcliffe',
'Location_Wollongong',
'Location_Woomera',
'WindGustDir_E',
'WindGustDir_ENE',
'WindGustDir_ESE',
'WindGustDir_N',
'WindGustDir_NE',
'WindGustDir_NNE',
'WindGustDir_NNW',
'WindGustDir_NW',
'WindGustDir_S',
'WindGustDir_SE',
'WindGustDir_SSE',
'WindGustDir_SSW',
'WindGustDir_SW',
'WindGustDir_W',
'WindGustDir_WNW',
'WindGustDir_WSW',
'WindGustDir_nan',
'WindDir9am_E',
'WindDir9am_ENE',
'WindDir9am_ESE',
'WindDir9am_N',
'WindDir9am_NE',
'WindDir9am_NNE',
'WindDir9am_NNW',
'WindDir9am_NW',
'WindDir9am_S',
'WindDir9am_SE',
'WindDir9am_SSE',
'WindDir9am_SSW',
'WindDir9am_SW',
'WindDir9am_W',
'WindDir9am_WNW',
'WindDir9am_WSW',
'WindDir9am_nan',
'WindDir3pm_E',
'WindDir3pm_ENE',
'WindDir3pm_ESE',
'WindDir3pm_N',
'WindDir3pm_NE',
'WindDir3pm_NNE',
'WindDir3pm_NNW',
'WindDir3pm_NW',
'WindDir3pm_S',
'WindDir3pm_SE',
'WindDir3pm_SSE',
'WindDir3pm_SSW',
'WindDir3pm_SW',
'WindDir3pm_W',
'WindDir3pm_WNW',
'WindDir3pm_WSW',
'WindDir3pm_nan',
'RainToday_No',
'RainToday_Yes']
```

```
In [9]: X_train = train_inputs[numeric_cols]
X_val = val_inputs[numeric_cols]
X_test = test_inputs[numeric_cols]
```

```
In [10]: X_val
```

Out[10]:

	MinTemp	MaxTemp	Rainfall	Evaporation	Sunshine	WindGustSpeed	WindSpeed9am	WindSp
<b>23480</b>	0.595855	0.504082	0.000959	0.082759	0.517483	0.254545	0.292308	
<b>77238</b>	0.461140	0.412245	0.004796	0.003448	0.391608	0.290909	0.261538	
<b>20541</b>	0.712435	0.867347	0.000000	0.094322	0.534317	0.572727	0.461538	
<b>57419</b>	0.251295	0.226531	0.000000	0.094322	0.534317	0.218182	0.338462	
<b>44800</b>	0.567358	0.459184	0.254197	0.094322	0.534317	0.236364	0.200000	
...	...	...	...	...	...	...	...	...
<b>2279</b>	0.321244	0.365306	0.000959	0.094322	0.534317	0.118182	0.000000	
<b>5429</b>	0.466321	0.673469	0.000000	0.094322	0.534317	0.154545	0.030769	
<b>83338</b>	0.256477	0.381633	0.000000	0.031034	0.363636	0.327273	0.061538	
<b>72707</b>	0.305699	0.371429	0.006715	0.096552	0.076923	0.372727	0.138462	
<b>54548</b>	0.582902	0.608163	0.000000	0.094322	0.534317	0.472727	0.292308	

1700 rows × 118 columns

## Training

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

```
In [11]: from sklearn.tree import DecisionTreeClassifier
model = DecisionTreeClassifier(random_state=42)
model.fit(X_train, train_targets)

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

Out[11]: `DecisionTreeClassifier(random_state=42)`

## Evaluation

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

```
In [12]: from sklearn.metrics import accuracy_score, confusion_matrix
train_preds = model.predict(X_train)
train_preds
```

Out[12]: `array(['No', 'No', 'No', ..., 'No', 'No', 'No'], dtype=object)`

```
In [13]: # returns the count of unique values in the `train_preds` variable.
# It creates a pandas Series object where the unique values in `train_preds` are the index and
pd.value_counts(train_preds)
```

```
Out[13]: No      7599
Yes     2189
dtype: int64
```

The decision tree also returns probabilities for each prediction.

```
In [14]: train_probability = model.predict_proba(X_train)
train_probability
```

```
Out[14]: 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.

```
In [15]: accuracy_score(train_targets, train_preds)
```

```
Out[15]: 1.0
```

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

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

```
In [16]: model.score(X_val, val_targets)
```

```
Out[16]: 0.8058823529411765
```

Although the training accuracy is 100%, the accuracy on the validation set is just about 80%, which is only marginally better than chance.

```
In [17]: val_targets.value_counts() / len(val_targets)
```

```
Out[17]: No      0.794118
Yes     0.205882
Name: RainTomorrow, dtype: float64
```

It appears that the model has learned the training examples perfect, and doesn't generalize well to previously unseen data. This 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.

```
In [18]: from sklearn.tree import plot_tree, export_text
import matplotlib.pyplot as plt
import seaborn as sns
import matplotlib
%matplotlib inline
```

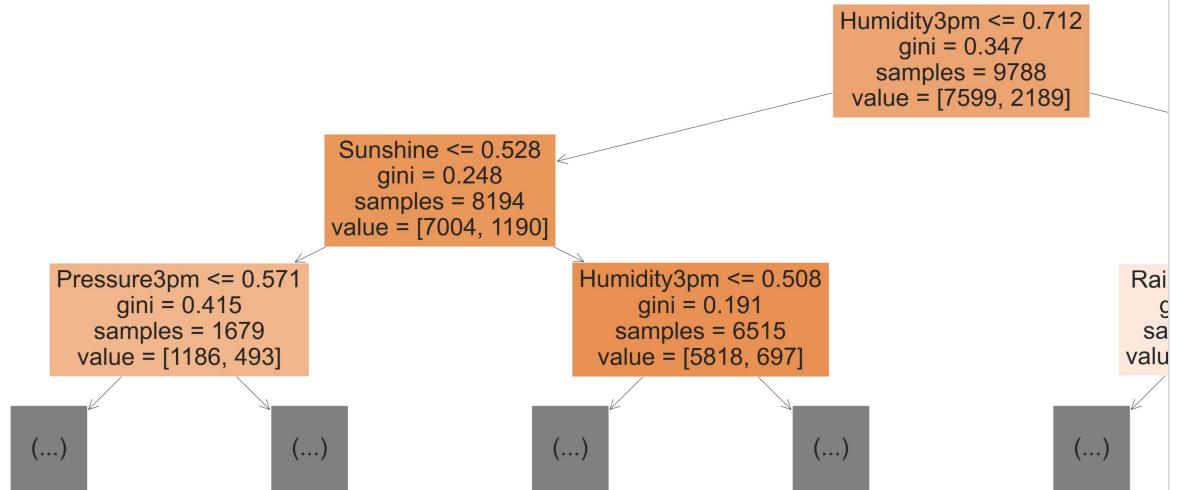
```
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 150)
sns.set_style('darkgrid')
matplotlib.rcParams['font.size'] = 14
matplotlib.rcParams['figure.figsize'] = (10, 6)
matplotlib.rcParams['figure.facecolor'] = '#00000000'
```

In [19]:

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

Out[19]:

```
[Text(0.5, 0.875, 'Humidity3pm <= 0.712\ngini = 0.347\nsamples = 9788\nvalue = [7599, 2189]'),
Text(0.25, 0.625, 'Sunshine <= 0.528\ngini = 0.248\nsamples = 8194\nvalue = [7004, 1190]'),
Text(0.125, 0.375, 'Pressure3pm <= 0.571\ngini = 0.415\nsamples = 1679\nvalue = [1186, 493]'),
Text(0.0625, 0.125, '\n (...) \n'),
Text(0.1875, 0.125, '\n (...) \n'),
Text(0.375, 0.375, 'Humidity3pm <= 0.508\ngini = 0.191\nsamples = 6515\nvalue = [5818, 697]'),
Text(0.3125, 0.125, '\n (...) \n'),
Text(0.4375, 0.125, '\n (...) \n'),
Text(0.75, 0.625, 'Humidity3pm <= 0.813\ngini = 0.468\nsamples = 1594\nvalue = [595, 999]'),
Text(0.625, 0.375, 'Rainfall <= 0.013\ngini = 0.496\nsamples = 830\nvalue = [454, 376]'),
Text(0.5625, 0.125, '\n (...) \n'),
Text(0.6875, 0.125, '\n (...) \n'),
Text(0.875, 0.375, 'Temp9am <= 0.305\ngini = 0.301\nsamples = 764\nvalue = [141, 623]'),
Text(0.8125, 0.125, '\n (...) \n'),
Text(0.9375, 0.125, '\n (...) \n')]
```



Can you see how the model classifies a given input as a series of decisions? The tree is truncated here, but follows "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. "Gini" is a selection criteria or tree splitting criteria. This is the loss function splitting the data, and at what point the column should be split. "A lower Gini index indicates a better split(resulting in each side) has a Gini index of 0. If there is 50-50 split then the criteria is least preferred by the model itself because it does not help in classification."

For a mathematical discussion of the Gini Index, watch this video: <https://www.youtube.com/watch?v=-W0Dnx>

## Gini Index

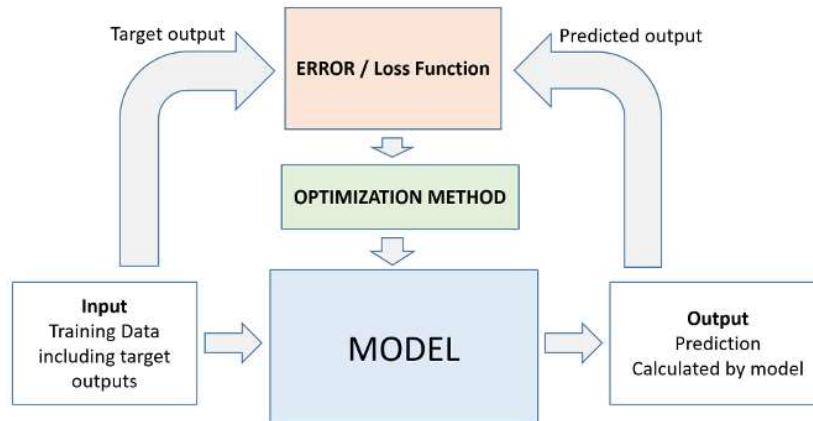
$$I_G = 1 - \sum^c n_i^2$$

$$\sum_{j=1}^J p_j$$

$p_j$ : 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 partitions the data into two portions. In practice, however, it's very inefficient to check all possible splits, so the model uses a heuristic to find the best split.

The iterative approach of the machine learning workflow in the case of a decision tree involves growing the tree by iteratively splitting the data into smaller and smaller subsets until a stopping criterion is met.



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

```
In [20]: model.tree_.max_depth
```

```
Out[20]: 30
```

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

```
In [21]: tree_text = export_text(model, max_depth=10, feature_names=list(X_train.columns))
print(tree_text[:5000])
```

```

--- Humidity3pm <= 0.71
|--- Sunshine <= 0.53
|   |--- Pressure3pm <= 0.57
|   |--- Humidity3pm <= 0.56
|   |   |--- WindGustSpeed <= 0.33
|   |   |--- Location_Albany <= 0.50
|   |   |   |--- Sunshine <= 0.00
|   |   |   |   |--- WindDir3pm_N <= 0.50
|   |   |   |   |--- class: Yes
|   |   |   |   |--- WindDir3pm_N > 0.50
|   |   |   |   |--- class: No
|   |   |--- Sunshine > 0.00
|   |   |   |--- WindDir3pm_nan <= 0.50
|   |   |   |   |--- WindSpeed9am <= 0.05
|   |   |   |   |   |--- WindDir3pm_NW <= 0.50
|   |   |   |   |   |--- class: Yes
|   |   |   |   |   |--- WindDir3pm_NW > 0.50
|   |   |   |   |   |--- class: No
|   |   |   |   |--- WindSpeed9am > 0.05
|   |   |   |   |   |--- WindGustDir_NW <= 0.50
|   |   |   |   |   |--- Pressure3pm <= 0.41
|   |   |   |   |   |   |--- truncated branch of depth 3
|   |   |   |   |   |--- Pressure3pm > 0.41
  
```

```
|   |   |   |   |   |   |   |--- truncated branch of depth 8
|   |   |   |   |   |   |   |--- WindGustDir_NW >  0.50
|   |   |   |   |   |   |   |--- Temp9am <= 0.47
|   |   |   |   |   |   |   |--- class: Yes
|   |   |   |   |   |   |--- Temp9am >  0.47
|   |   |   |   |   |   |--- class: No
|   |   |   |   |   |--- WindDir3pm_nan >  0.50
|   |   |   |   |   |--- class: Yes
|   |   |   |   |--- Location_Albany >  0.50
|   |   |   |   |--- WindSpeed9am <= 0.12
|   |   |   |   |   |--- class: No
|   |   |   |   |--- WindSpeed9am >  0.12
|   |   |   |   |   |--- class: Yes
|   |   |   |--- WindGustSpeed >  0.33
|   |   |   |--- Location_Portland <= 0.50
|   |   |   |--- Pressure9am <= 0.61
|   |   |   |--- Sunshine <= 0.01
|   |   |   |   |--- class: Yes
|   |   |   |--- Sunshine >  0.01
|   |   |   |--- Temp3pm <= 0.77
|   |   |   |--- MaxTemp <= 0.75
|   |   |   |--- Location_Dartmoor <= 0.50
|   |   |   |   |--- truncated branch of depth 15
|   |   |   |--- Location_Dartmoor >  0.50
|   |   |   |   |--- class: Yes
|   |   |   |--- MaxTemp >  0.75
|   |   |   |--- WindSpeed3pm <= 0.20
|   |   |   |   |--- class: No
|   |   |   |--- WindSpeed3pm >  0.20
|   |   |   |   |--- class: Yes
|   |   |   |--- Temp3pm >  0.77
|   |   |   |   |--- class: No
|   |   |   |--- Pressure9am >  0.61
|   |   |   |   |--- class: Yes
|   |   |   |--- Location_Portland >  0.50
|   |   |   |--- class: Yes
|--- Humidity3pm >  0.56
|--- WindGustSpeed <= 0.54
|--- Rainfall <= 0.02
|--- Pressure3pm <= 0.36
|--- MinTemp <= 0.39
|--- Pressure9am <= 0.39
|   |--- class: No
|--- Pressure9am >  0.39
|--- WindDir9am_NW <= 0.50
|   |--- class: Yes
|--- WindDir9am_NW >  0.50
|   |--- class: No
|--- MinTemp >  0.39
|--- WindSpeed3pm <= 0.16
|   |--- class: No
|--- WindSpeed3pm >  0.16
|   |--- class: Yes
|--- Pressure3pm >  0.36
|--- WindGustDir_NW <= 0.50
|--- WindGustDir_WNW <= 0.50
|--- MinTemp <= 0.52
|--- MinTemp <= 0.50
|   |--- truncated branch of depth 7
|--- MinTemp >  0.50
|   |--- truncated branch of depth 2
|--- MinTemp >  0.52
|--- WindGustSpeed <= 0.33
|   |--- truncated branch of depth 10
|--- WindGustSpeed >  0.33
|   |--- truncated branch of depth 5
|--- WindGustDir_WNW >  0.50
```

## Why Overfitting or 100% training accuracy?

The training accuracy is 100% because the decision tree has (our model) has literally memorized the entire training set.

And when it sees any example other than training or which doesn't fit exactly to the training, it tries to categorize it using DT. That may or may not end up really well because it's going to ultimately boil down to a specific training example.

## Feature Importance

Based on the gini index computations, a decision tree assigns an "importance" value to each feature. These values range from 0.0 to 1.0.

In [22]:

```
X_train.columns
```

Out[22]:

```
Index(['MinTemp', 'MaxTemp', 'Rainfall', 'Evaporation', 'Sunshine',
       'WindGustSpeed', 'WindSpeed9am', 'WindSpeed3pm', 'Humidity9am',
       'Humidity3pm',
       ...
       'WindDir3pm_SE', 'WindDir3pm_SSE', 'WindDir3pm_SSW', 'WindDir3pm_SW',
       'WindDir3pm_W', 'WindDir3pm_WNW', 'WindDir3pm_WSW', 'WindDir3pm_nan',
       'RainToday_No', 'RainToday_Yes'],
      dtype='object', length=118)
```

In [23]:

```
model.feature_importances_
```

Out[23]:

```
array([3.49257345e-02, 3.07937151e-02, 4.22748141e-02, 1.86728347e-02,
       5.44097237e-02, 5.15867301e-02, 2.12312331e-02, 2.87536812e-02,
       3.68257188e-02, 2.64559367e-01, 4.61709226e-02, 6.74395647e-02,
       1.73926170e-02, 1.25463339e-02, 3.30940709e-02, 2.81314210e-02,
       2.53636257e-03, 4.88594907e-03, 8.85196443e-05, 5.58347395e-04,
       2.17801237e-03, 1.72048150e-03, 1.79704909e-03, 8.80837628e-04,
       1.85616582e-03, 6.79028712e-04, 5.48664767e-04, 1.60682190e-03,
       3.36763438e-03, 7.65438990e-04, 1.33296623e-03, 1.76337264e-03,
       0.00000000e+00, 2.89718184e-03, 1.15398601e-03, 1.04335506e-03,
       1.42614666e-04, 9.47687991e-04, 8.43648412e-05, 2.30446882e-03,
       6.13116680e-04, 7.75828463e-04, 8.40861880e-04, 3.19148702e-03,
       2.03150657e-03, 1.95633156e-03, 8.01171026e-04, 5.46423397e-04,
       0.00000000e+00, 4.47298844e-03, 1.36814558e-03, 0.00000000e+00,
       1.17777289e-03, 1.32691363e-03, 1.41449744e-03, 1.08994330e-03,
       1.99200741e-03, 5.05498582e-04, 7.28326586e-04, 1.32816124e-03,
       1.86866937e-03, 6.17388159e-04, 1.81192841e-03, 2.20911476e-03,
       0.00000000e+00, 1.83791100e-03, 1.03845236e-03, 7.52022031e-04,
       4.19639250e-03, 2.11818535e-03, 1.83924445e-03, 2.63761456e-03,
       6.51501283e-03, 4.08796835e-03, 4.29656774e-03, 1.08977592e-03,
       1.43907060e-03, 1.91287844e-03, 3.05273200e-03, 3.52894824e-03,
       5.20036521e-03, 7.02232411e-04, 2.10623085e-03, 2.04384088e-03,
       0.00000000e+00, 5.15174436e-03, 4.10503777e-03, 4.36257790e-03,
       5.36214897e-03, 4.57599897e-03, 1.75689777e-03, 1.62553252e-03,
       1.59417504e-03, 4.05762243e-03, 4.86851289e-03, 2.09933276e-03,
       2.76186580e-03, 1.51936939e-03, 1.38712067e-03, 1.53936409e-03,
       3.18368035e-03, 9.87483603e-04, 3.62194595e-03, 2.82413190e-03,
       1.44448964e-03, 4.96895742e-03, 4.35381001e-03, 2.64299215e-03,
       2.84420435e-03, 1.55220559e-03, 2.40836903e-03, 1.29944045e-03,
       4.59302940e-03, 1.16876609e-03, 3.88290293e-03, 1.73375026e-03,
       2.02471991e-03, 6.86496843e-04])
```

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

In [24]:

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

```
}).sort_values('importance', ascending=False)
```

In [25]:  
importance\_df

Out[25]:

	feature	importance
9	Humidity3pm	0.264559
11	Pressure3pm	0.067440
4	Sunshine	0.054410
5	WindGustSpeed	0.051587
10	Pressure9am	0.046171
2	Rainfall	0.042275
8	Humidity9am	0.036826
0	MinTemp	0.034926
14	Temp9am	0.033094
1	MaxTemp	0.030794
7	WindSpeed3pm	0.028754
15	Temp3pm	0.028131
6	WindSpeed9am	0.021231
3	Evaporation	0.018673
12	Cloud9am	0.017393
13	Cloud3pm	0.012546
72	WindGustDir_NW	0.006515
88	WindDir9am_NNW	0.005362
80	WindGustDir_WSW	0.005200
85	WindDir9am_N	0.005152
105	WindDir3pm_NNW	0.004969
17	Location_Albany	0.004886
94	WindDir9am_SW	0.004869
112	WindDir3pm_W	0.004593
89	WindDir9am_NW	0.004576
49	Location_Portland	0.004473
87	WindDir9am_NNE	0.004363
106	WindDir3pm_NW	0.004354
74	WindGustDir_SE	0.004297
68	WindGustDir_N	0.004196
86	WindDir9am_NE	0.004105
73	WindGustDir_S	0.004088

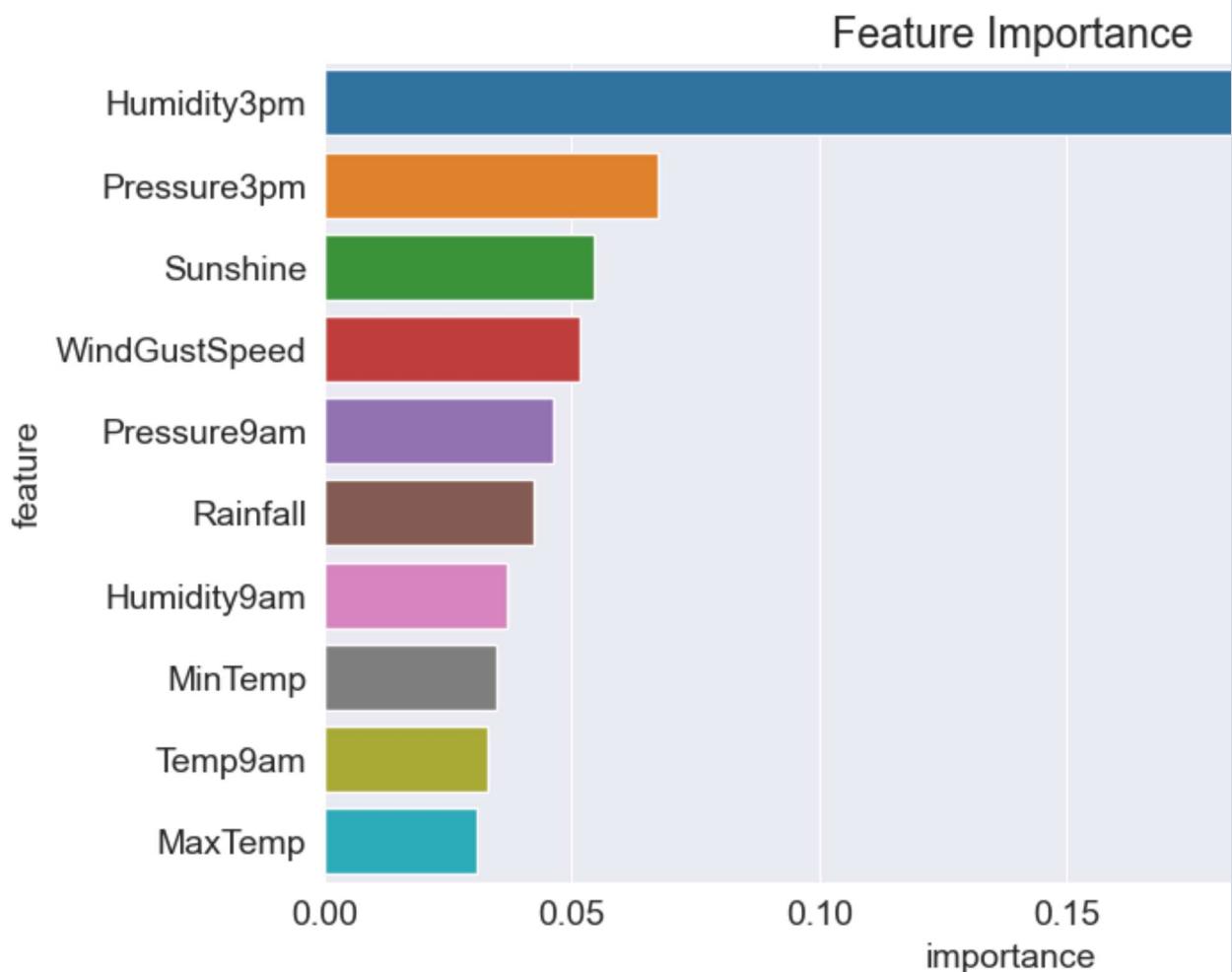
<b>93</b>	WindDir9am_SSW	0.004058
<b>114</b>	WindDir3pm_WSW	0.003883
<b>102</b>	WindDir3pm_N	0.003622
<b>79</b>	WindGustDir_WNW	0.003529
<b>28</b>	Location_Dartmoor	0.003368
<b>43</b>	Location_NorfolkIsland	0.003191
<b>100</b>	WindDir3pm_ENE	0.003184
<b>78</b>	WindGustDir_W	0.003053
<b>33</b>	Location_Launceston	0.002897
<b>108</b>	WindDir3pm_SE	0.002844
<b>103</b>	WindDir3pm_NE	0.002824
<b>96</b>	WindDir9am_WNW	0.002762
<b>107</b>	WindDir3pm_S	0.002643
<b>71</b>	WindGustDir_NNW	0.002638
<b>16</b>	Location_Adelaide	0.002536
<b>110</b>	WindDir3pm_SSW	0.002408
<b>39</b>	Location_MountGinini	0.002304
<b>63</b>	Location_Wollongong	0.002209
<b>20</b>	Location_BadgerysCreek	0.002178
<b>69</b>	WindGustDir_NE	0.002118
<b>82</b>	WindDir9am_E	0.002106
<b>95</b>	WindDir9am_W	0.002099
<b>83</b>	WindDir9am_ENE	0.002044
<b>44</b>	Location_Nuriootpa	0.002032
<b>116</b>	RainToday_No	0.002025
<b>56</b>	Location_Tuggeranong	0.001992
<b>45</b>	Location_PearceRAAF	0.001956
<b>77</b>	WindGustDir_SW	0.001913
<b>60</b>	Location_Watsonia	0.001869
<b>24</b>	Location_Cairns	0.001856
<b>70</b>	WindGustDir_NNE	0.001839
<b>65</b>	WindGustDir_E	0.001838
<b>62</b>	Location_Witchcliffe	0.001812
<b>22</b>	Location_Bendigo	0.001797
<b>31</b>	Location_Hobart	0.001763
<b>90</b>	WindDir9am_S	0.001757
<b>115</b>	WindDir3pm nan	0.001734

<b>21</b>	Location_Ballarat	0.001720
<b>91</b>	WindDir9am_SE	0.001626
<b>27</b>	Location_CoffsHarbour	0.001607
<b>92</b>	WindDir9am_SSE	0.001594
<b>109</b>	WindDir3pm_SSE	0.001552
<b>99</b>	WindDir3pm_E	0.001539
<b>97</b>	WindDir9am_WSW	0.001519
<b>104</b>	WindDir3pm_NNE	0.001444
<b>76</b>	WindGustDir_SSW	0.001439
<b>54</b>	Location_SydneyAirport	0.001414
<b>98</b>	WindDir9am_nan	0.001387
<b>50</b>	Location_Richmond	0.001368
<b>30</b>	Location_GoldCoast	0.001333
<b>59</b>	Location_Walpole	0.001328
<b>53</b>	Location_Sydney	0.001327
<b>111</b>	WindDir3pm_SW	0.001299
<b>52</b>	Location_SalmonGums	0.001178
<b>113</b>	WindDir3pm_WNW	0.001169
<b>34</b>	Location_Melbourne	0.001154
<b>55</b>	Location_Townsville	0.001090
<b>75</b>	WindGustDir_SSE	0.001090
<b>35</b>	Location_MelbourneAirport	0.001043
<b>66</b>	WindGustDir_ENE	0.001038
<b>101</b>	WindDir3pm_ESE	0.000987
<b>37</b>	Location_Moree	0.000948
<b>23</b>	Location_Brisbane	0.000881
<b>42</b>	Location_NorahHead	0.000841
<b>46</b>	Location_Penrith	0.000801
<b>41</b>	Location_Nhil	0.000776
<b>29</b>	Location_Darwin	0.000765
<b>67</b>	WindGustDir_ESE	0.000752
<b>58</b>	Location_WaggaWagga	0.000728
<b>81</b>	WindGustDir_nan	0.000702
<b>117</b>	RainToday_Yes	0.000686
<b>25</b>	Location_Canberra	0.000679
<b>61</b>	Location_Williamtown	0.000617

40	Location_Newcastle	0.000613
19	Location_AliceSprings	0.000558
26	Location_Cobar	0.000549
47	Location_Perth	0.000546
57	Location_Uluru	0.000505
36	Location_Mildura	0.000143
18	Location_Albury	0.000089
38	Location_MountGambier	0.000084
84	WindDir9am_ESE	0.000000
48	Location_PerthAirport	0.000000
32	Location_Katherine	0.000000
51	Location_Sale	0.000000
64	Location_Woomera	0.000000

In [26]:

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



## Hyperparameter Tuning and Overfitting

As we saw in the previous section, our decision tree classifier memorized all training examples, leading to a 100% better than a dumb baseline model. This phenomenon is called overfitting, and in this section, we'll look at some ways to prevent it.

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

- `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.

```
In [27]: model = DecisionTreeClassifier(max_depth= 3, random_state= 42)
model.fit(X_train, train_targets)
```

```
Out[27]: DecisionTreeClassifier(max_depth=3, random_state=42)
```

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

```
In [28]: print("Training Score: ", model.score(X_train, train_targets))
print("Validation Score: ", model.score(X_val, val_targets))
```

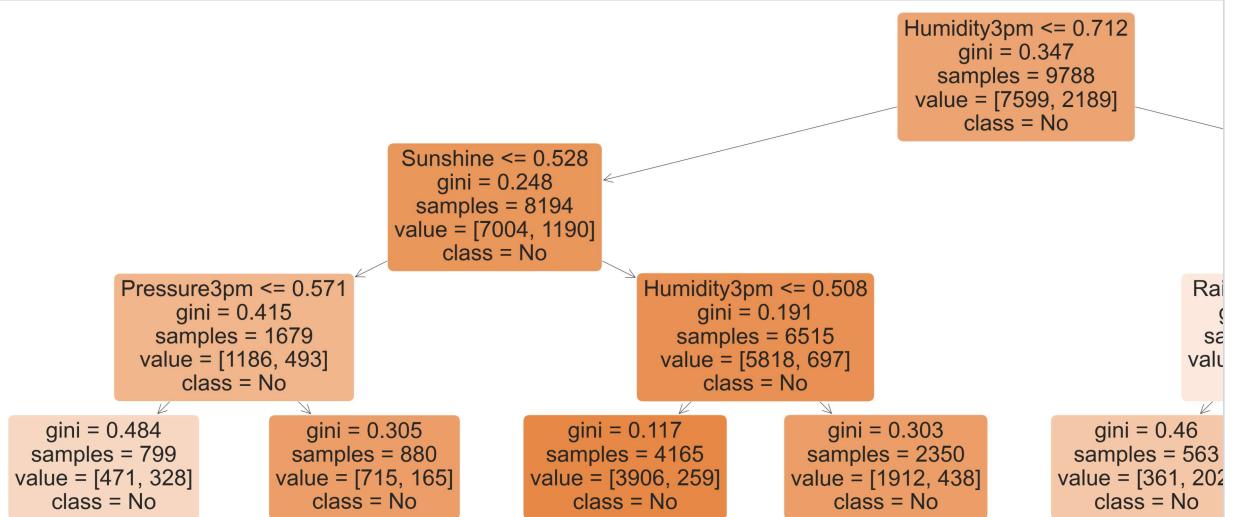
```
Training Score:  0.8338782182263996
Validation Score:  0.8470588235294118
```

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

```
In [29]: model.classes_
```

```
Out[29]: array(['No', 'Yes'], dtype=object)
```

```
In [30]: plt.figure(figsize=(80,20))
plot_tree(model, feature_names=X_train.columns, filled=True, rounded=True, class_names=model.classes_)
```



```
In [31]: print(export_text(model, feature_names=list(X_train.columns)))

--- Humidity3pm <= 0.71
|--- Sunshine <= 0.53
|   |--- Pressure3pm <= 0.57
|   |   |--- class: No
|   |--- Pressure3pm >  0.57
|   |   |--- class: No
|--- Sunshine >  0.53
|   |--- Humidity3pm <= 0.51
|   |   |--- class: No
|   |--- Humidity3pm >  0.51
|   |   |--- class: No
--- Humidity3pm >  0.71
|--- Humidity3pm <= 0.81
|   |--- Rainfall <= 0.01
|   |   |--- class: No
|   |--- Rainfall >  0.01
|   |   |--- class: Yes
|--- Humidity3pm >  0.81
|   |--- Temp9am <= 0.30
|   |   |--- class: Yes
|   |--- Temp9am >  0.30
|   |   |--- class: Yes
```

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

```
In [39]: def max_depth_error(md):
    model = DecisionTreeClassifier(max_depth=md, random_state=42)
    model.fit(X_train, train_targets)
    train_error = 1 - model.score(X_train, train_targets)
    validation_error = 1 - model.score(X_val, val_targets)
    return {'Max Depth': md, 'Training Error': train_error,
            'Validation Error': validation_error}
```

```
In [40]: error_df = pd.DataFrame([max_depth_error(md) for md in range (1,21)])
```

```
In [41]: error_df
```

```
Out[41]: Max Depth Training Error Validation Error
```

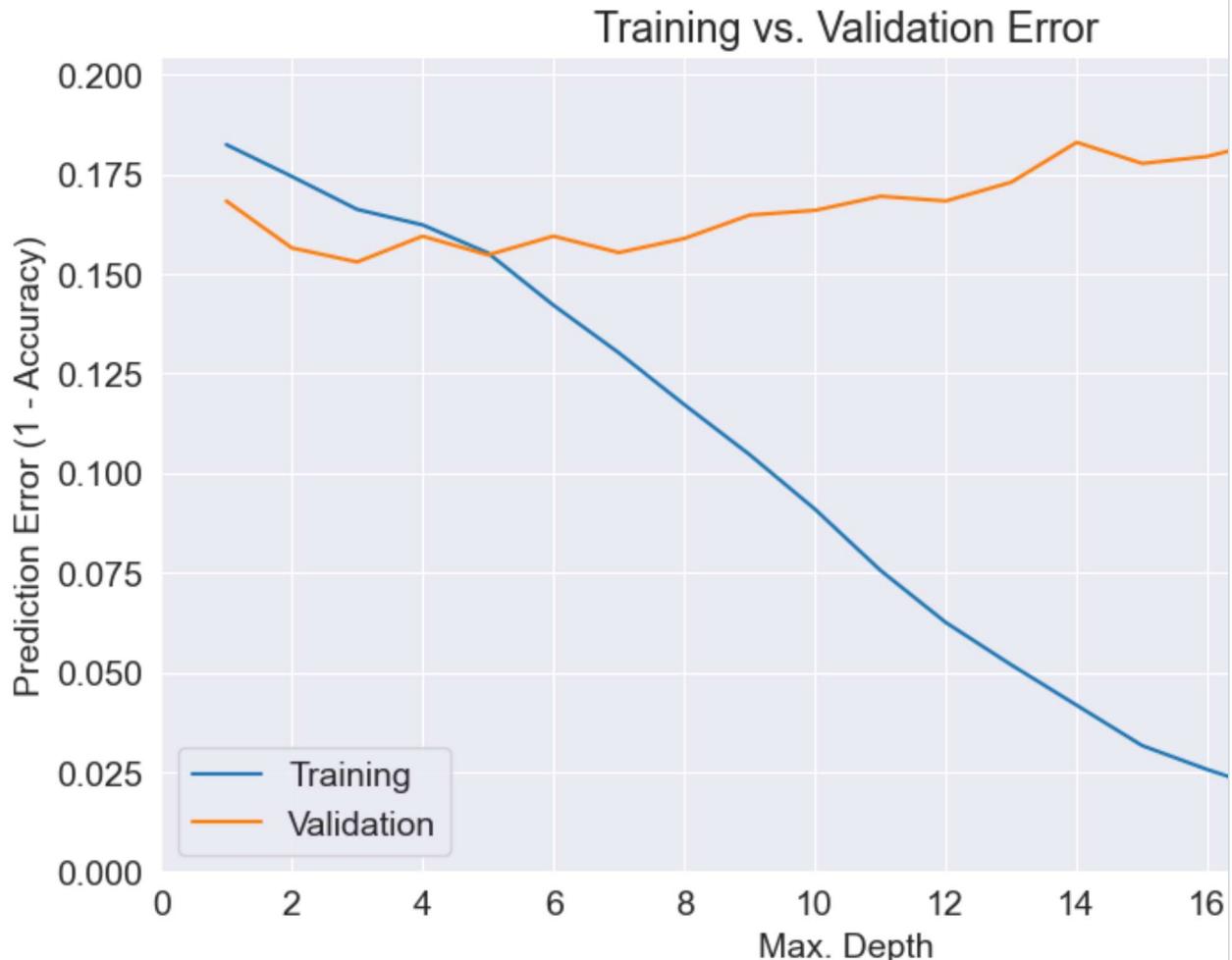
<b>0</b>	1	0.182366	0.168235
<b>1</b>	2	0.174397	0.156471
<b>2</b>	3	0.166122	0.152941
<b>3</b>	4	0.162239	0.159412
<b>4</b>	5	0.155190	0.154706
<b>5</b>	6	0.142113	0.159412
<b>6</b>	7	0.130159	0.155294
<b>7</b>	8	0.117184	0.158824
<b>8</b>	9	0.104618	0.164706
<b>9</b>	10	0.090928	0.165882
<b>10</b>	11	0.075603	0.160112

11	12	0.062526	0.168235
12	13	0.051900	0.172941
13	14	0.041786	0.182941
14	15	0.031671	0.177647
15	16	0.025644	0.179412
16	17	0.020331	0.183529
17	18	0.016347	0.192353
18	19	0.013282	0.183529
19	20	0.009195	0.195294

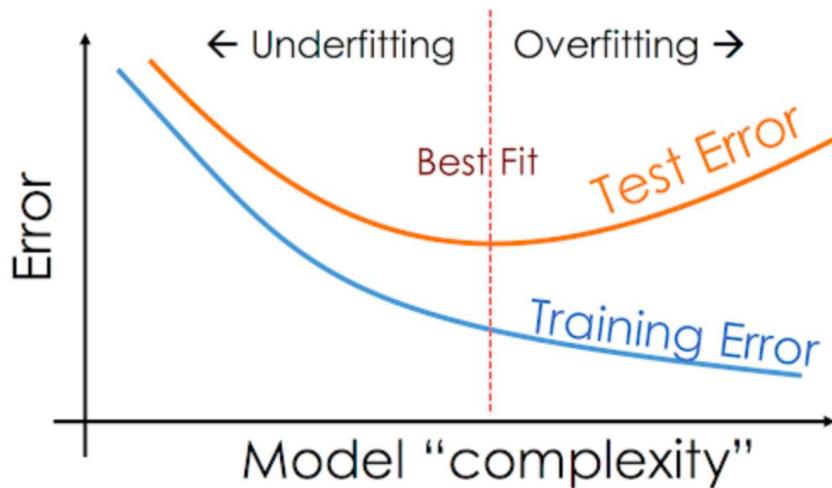
In [35]:

```
plt.figure()
plt.plot(error_df['Max Depth'], error_df['Training Error'])
plt.plot(error_df['Max Depth'], error_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'])
```

Out[35]: &lt;matplotlib.legend.Legend at 0x1ad38a69940&gt;



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

```
In [42]: model = DecisionTreeClassifier(max_depth=3, random_state= 42).fit(X_train, train_targets)
model.score(X_train, train_targets), model.score(X_val, val_targets)
```

```
Out[42]: (0.8338782182263996, 0.8470588235294118)
```

```
In [43]: model = DecisionTreeClassifier(max_depth=5, random_state= 42).fit(X_train, train_targets)
model.score(X_train, train_targets), model.score(X_val, val_targets)
```

```
Out[43]: (0.8448099713935431, 0.8452941176470589)
```

```
In [44]: model = DecisionTreeClassifier(max_depth=7, random_state= 42).fit(X_train, train_targets)
model.score(X_train, train_targets), model.score(X_val, val_targets)
```

```
Out[44]: (0.8698406211687781, 0.8447058823529412)
```

## max\_leaf\_nodes

Another way to control the size of complexity of a decision tree is to limit the number of leaf nodes. This allow

`max_leaf_nodes = 2^n`

example,  $2^7 = 128$

```
In [45]: model = DecisionTreeClassifier(max_leaf_nodes= 8, random_state= 42)
model.fit(X_train, train_targets)
```

```
Out[45]: DecisionTreeClassifier(max_leaf_nodes=8, random_state=42)
```

```
In [47]: model.score(X_train, train_targets), model.score(X_val, val_targets)
```

```
Out[47]: (0.8338782182263996, 0.8470588235294118)
```

```
In [50]: model.tree_.max_depth
```

```
Out[50]: 4
```

Notice that the model was able to achieve a greater depth of 4 for certain paths while keeping other paths short.

In [53]:

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

--- Humidity3pm <= 0.71
|--- Sunshine <= 0.53
|   |--- Pressure3pm <= 0.57
|   |   |--- class: No
|   |--- Pressure3pm > 0.57
|   |   |--- class: No
|--- Sunshine > 0.53
|   |--- Humidity3pm <= 0.51
|   |   |--- class: No
|   |--- Humidity3pm > 0.51
|       |--- WindGustSpeed <= 0.38
|           |--- class: No
|           |--- WindGustSpeed > 0.38
|               |--- class: No
--- Humidity3pm > 0.71
|--- Humidity3pm <= 0.81
|   |--- Rainfall <= 0.01
|   |   |--- class: No
|   |--- Rainfall > 0.01
|       |--- class: Yes
|--- Humidity3pm > 0.81
|   |--- class: Yes
```

Let's check the combination of `max_depth` and `max_leaf_nodes` that results in the highest validation accuracy.

In [54]:

```
model = DecisionTreeClassifier(max_depth=3, max_leaf_nodes=8, random_state= 42).fit(X_train, train_targets)
model.score(X_train, train_targets), model.score(X_val, val_targets)
```

Out[54]:

```
(0.8338782182263996, 0.8470588235294118)
```

In [55]:

```
model = DecisionTreeClassifier(max_depth=7, max_leaf_nodes= 128, random_state= 42).fit(X_train, train_targets)
model.score(X_train, train_targets), model.score(X_val, val_targets)
```

Out[55]:

```
(0.8698406211687781, 0.8435294117647059)
```

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>

## Training a Random Forest

While tuning the hyperparameters of a single decision tree may lead to some improvements, a much more effective approach is to train multiple decision trees simultaneously and average their predictions, resulting in a model 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 their predictions, the overall error is reduced, which is known as the "wisdom of the crowd".

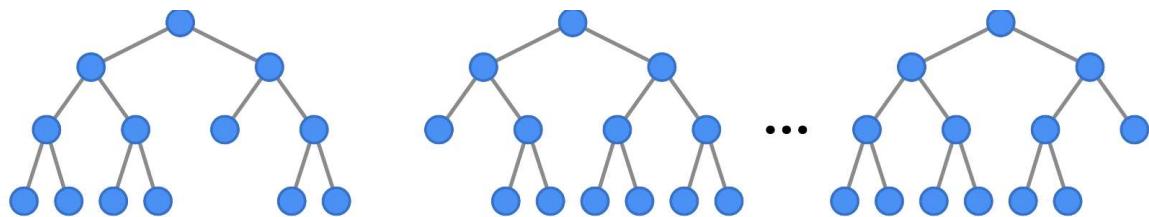
A random forest works by averaging/combing the results of several decision trees:

## EXAMPLES

Tree-1

Tree-2

Tree-n



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

In [56]:

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

`n_jobs` allows the random forest to use multiple parallel workers to train decision trees, and `random_state`:

In [57]:

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

Out[57]: `RandomForestClassifier(n_jobs=-1, random_state=42)`

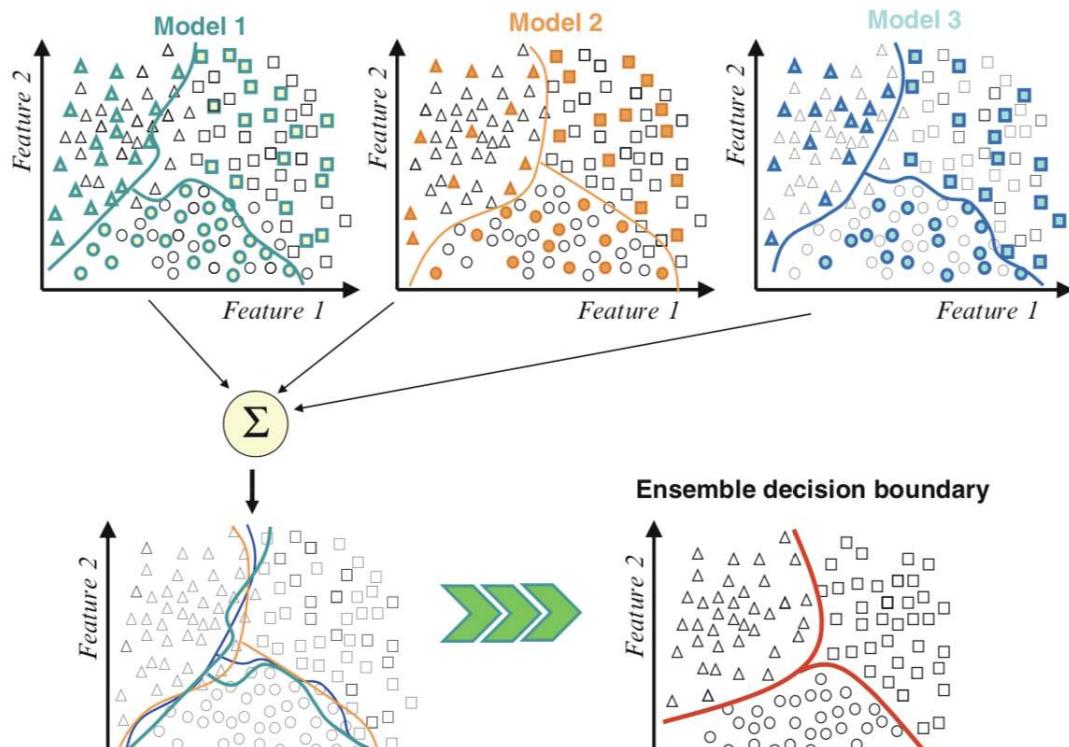
In [59]:

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

Out[59]: `(1.0, 0.8611764705882353)`

Once again, the training accuracy is almost 100%, but this time the validation accuracy is much better. In fact, if you see the power of random forests?

This general technique of combining the results of many models is called "ensembling", it works because most looks like visually:





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

```
In [60]: train_probs = model.predict_proba(X_train)
train_probs
```

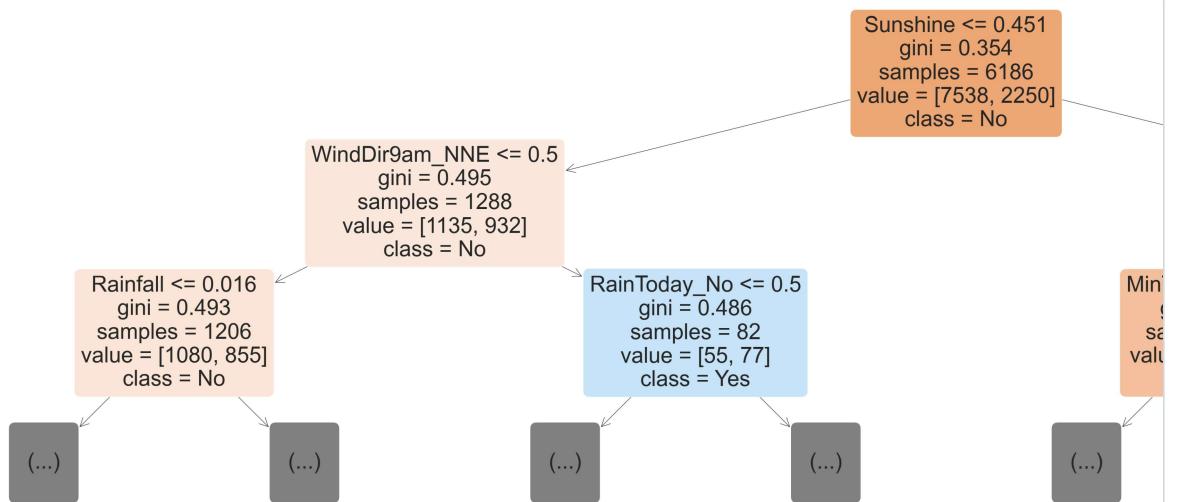
```
Out[60]: array([[0.87, 0.13],
   [0.88, 0.12],
   [0.98, 0.02],
   ...,
   [0.9 , 0.1 ],
   [0.96, 0.04],
   [0.98, 0.02]])
```

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

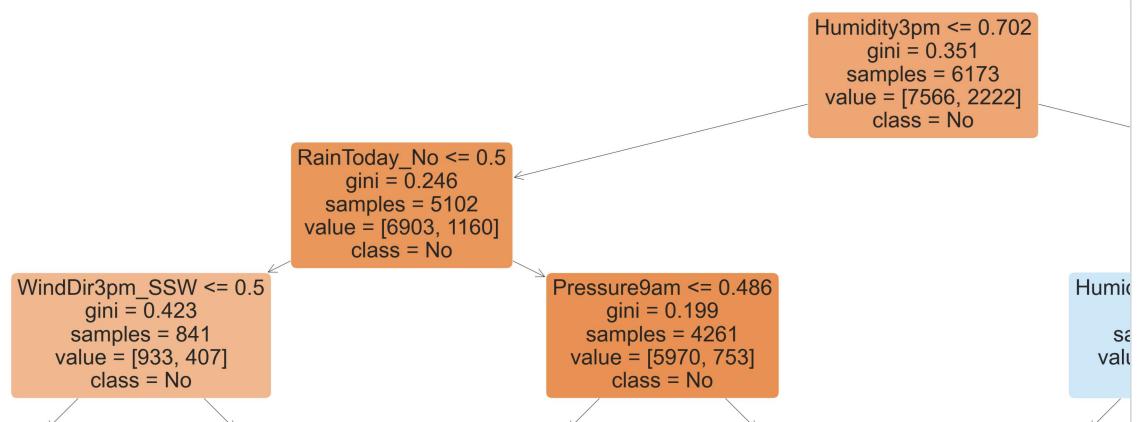
```
In [62]: model.estimators_[0]
```

```
Out[62]: DecisionTreeClassifier(max_features='auto', random_state=1608637542)
```

```
In [63]: plt.figure(figsize=(80,20))
plot_tree(model.estimators_[0], max_depth=2, feature_names=X_train.columns, filled=True, rounded=True)
```



```
In [64]: plt.figure(figsize=(80,20))
plot_tree(model.estimators_[15], max_depth=2, feature_names=X_train.columns, filled=True, rounded=True)
```



(...)

(...)

(...)

(...)

(...)

Just like decision tree, random forests also assign an "importance" to each feature, by combining the importan

In [65]:

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

importance_df
```

Out[65]:

	feature	importance
9	Humidity3pm	0.123956
4	Sunshine	0.057400
11	Pressure3pm	0.051902
10	Pressure9am	0.049768
2	Rainfall	0.049051
8	Humidity9am	0.048557
5	WindGustSpeed	0.046645
15	Temp3pm	0.043211
1	MaxTemp	0.041805
0	MinTemp	0.040839
14	Temp9am	0.038864
13	Cloud3pm	0.036358
7	WindSpeed3pm	0.031184
6	WindSpeed9am	0.029869
3	Evaporation	0.026449
116	RainToday_No	0.023681
12	Cloud9am	0.023680
117	RainToday_Yes	0.014562
85	WindDir9am_N	0.005018
105	WindDir3pm_NNW	0.004133
72	WindGustDir_NW	0.004060
106	WindDir3pm_NW	0.003978
89	WindDir9am_NW	0.003758
88	WindDir9am_NNW	0.003718
62	Location_Witchcliffe	0.003691
96	WindDir9am_WNW	0.003687
87	WindDir9am_NNE	0.003672

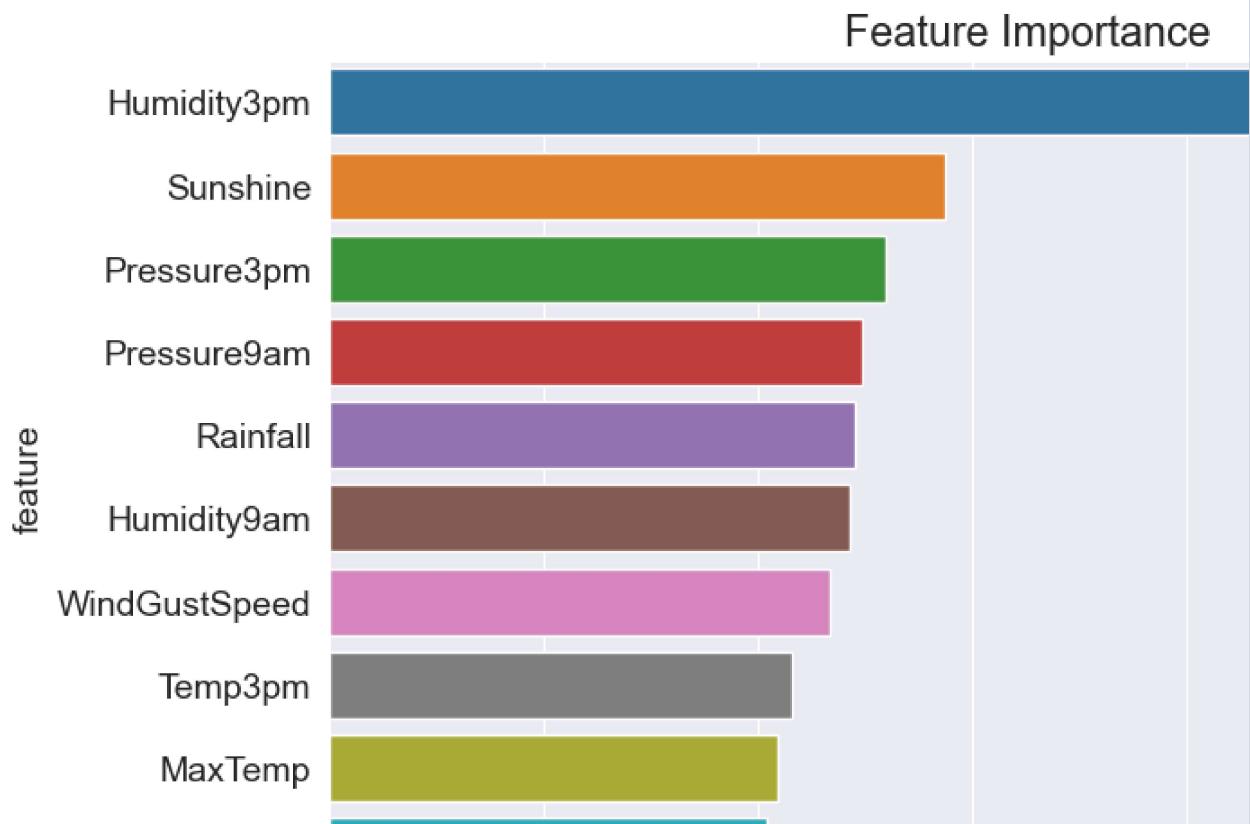
71	WindGustDir_NNW	0.003550
95	WindDir9am_W	0.003440
80	WindGustDir_WSW	0.003407
102	WindDir3pm_N	0.003365
114	WindDir3pm_WSW	0.003365
76	WindGustDir_SSW	0.003347
78	WindGustDir_W	0.003328
107	WindDir3pm_S	0.003321
112	WindDir3pm_W	0.003317
73	WindGustDir_S	0.003292
94	WindDir9am_SW	0.003245
108	WindDir3pm_SE	0.003130
90	WindDir9am_S	0.003095
104	WindDir3pm_NNE	0.003080
79	WindGustDir_WNW	0.003021
113	WindDir3pm_WNW	0.003013
70	WindGustDir_NNE	0.002973
81	WindGustDir_nan	0.002971
111	WindDir3pm_SW	0.002965
77	WindGustDir_SW	0.002887
109	WindDir3pm_SSE	0.002885
74	WindGustDir_SE	0.002799
93	WindDir9am_SSW	0.002797
103	WindDir3pm_NE	0.002753
101	WindDir3pm_ESE	0.002750
68	WindGustDir_N	0.002729
86	WindDir9am_NE	0.002715
110	WindDir3pm_SSW	0.002697
98	WindDir9am_nan	0.002638
97	WindDir9am_WSW	0.002622
75	WindGustDir_SSE	0.002577
92	WindDir9am_SSE	0.002479
91	WindDir9am_SE	0.002362
69	WindGustDir_NE	0.002343
82	WindDir9am_E	0.002319
66	WindGustDir_ENE	0.002307
100	WindDir3pm_FNF	0.002286

99	WindDir3pm_E	0.002218
65	WindGustDir_E	0.002216
83	WindDir9am_ENE	0.002165
17	Location_Albany	0.002146
54	Location_SydneyAirport	0.002106
49	Location_Portland	0.002066
53	Location_Sydney	0.002023
27	Location_CoffsHarbour	0.001928
56	Location_Tuggeranong	0.001890
33	Location_Launceston	0.001840
28	Location_Dartmoor	0.001833
31	Location_Hobart	0.001829
38	Location_MountGambier	0.001818
39	Location_MountGinini	0.001803
59	Location_Walpole	0.001799
67	WindGustDir_ESE	0.001783
60	Location_Watsonia	0.001673
25	Location_Canberra	0.001635
84	WindDir9am_ESE	0.001616
43	Location_NorfolkIsland	0.001609
47	Location_Perth	0.001608
51	Location_Sale	0.001600
30	Location_GoldCoast	0.001575
63	Location_Wollongong	0.001570
16	Location_Adelaide	0.001520
115	WindDir3pm_nan	0.001513
20	Location_BadgerysCreek	0.001486
23	Location_Brisbane	0.001481
21	Location_Ballarat	0.001477
61	Location_Williamtown	0.001456
35	Location_MelbourneAirport	0.001371
36	Location_Mildura	0.001361
24	Location_Cairns	0.001338
34	Location_Melbourne	0.001328
52	Location_SalmonGums	0.001311
48	Location_PerthAirport	0.001298

22	Location_Bendigo	0.001283
50	Location_Richmond	0.001249
45	Location_PearceRAAF	0.001235
18	Location_Albury	0.001214
40	Location_Newcastle	0.001204
58	Location_WaggaWagga	0.001189
37	Location_Moree	0.001175
42	Location_NorahHead	0.001149
44	Location_Nuriootpa	0.001005
29	Location_Darwin	0.000981
46	Location_Penrith	0.000886
64	Location_Woomera	0.000829
26	Location_Cobar	0.000780
41	Location_Nhil	0.000764
55	Location_Townsville	0.000748
19	Location_AliceSprings	0.000628
32	Location_Katherine	0.000392
57	Location_Uluru	0.000336

In [68]:

```
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.

## Hyperparameter Tuning with Random Forests

Just like decision trees, random forests also have several hyperparameters. In fact many of these hyperparameters are shared with decision trees.

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

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

```
In [69]: base_model = RandomForestClassifier(random_state=42, n_jobs=-1).fit(X_train, train_targets)
base_train_accuracy = base_model.score(X_train, train_targets)
base_val_accuracy = base_model.score(X_val, val_targets)
base_accuracies = base_train_accuracy, base_val_accuracy
base_accuracies
```

```
Out[69]: (1.0, 0.8611764705882353)
```

We can use this as a benchmark for hyperparameter tuning.

### n\_estimators

This argument controls the number of decision trees in the random forest. The default value is 100. For larger datasets, rule of thumb is to have at least 100 estimators. For smaller datasets, try to have as few estimators as needed.

Note: Randomness helps reduce overfitting. More randomness, less overfitting.

#### 10 estimators

```
In [70]: model = RandomForestClassifier(random_state= 42, n_jobs= -1, n_estimators=10)
model.fit(X_train, train_targets)
```

```
Out[70]: RandomForestClassifier(n_estimators=10, n_jobs=-1, random_state=42)
```

```
In [73]: print(base_accuracies)
model.score(X_train, train_targets), model.score(X_val, val_targets)
```

```
(1.0, 0.8611764705882353)
```

```
Out[73]: (0.9871270944013077, 0.8470588235294118)
```

#### 500 estimators

```
In [74]: model = RandomForestClassifier(random_state= 42, n_jobs= -1, n_estimators=500)
model.fit(X_train, train_targets)
```

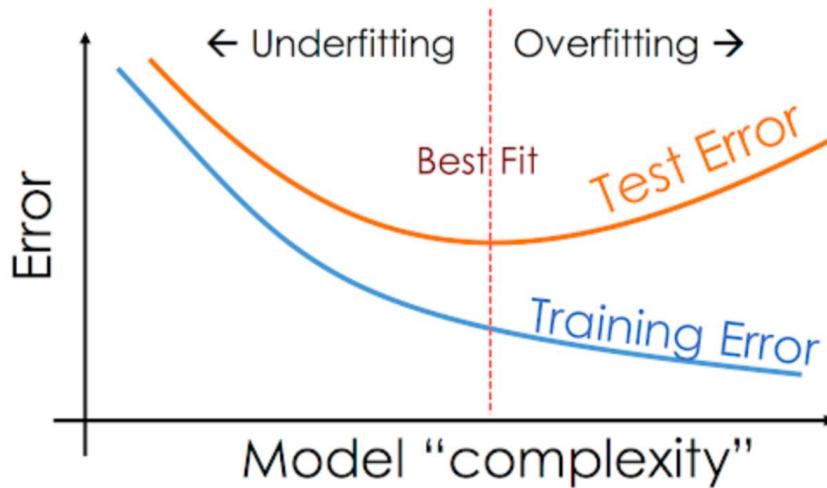
```
Out[74]: RandomForestClassifier(n_estimators=500, n_jobs=-1, random_state=42)
```

```
In [75]: print(base_accuracies)
model.score(X_train, train_targets), model.score(X_val, val_targets)

(1.0, 0.8611764705882353)
Out[75]: (1.0, 0.8641176470588235)
```

## 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 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.

```
In [76]: def test_params(**params):
    model = RandomForestClassifier(random_state=42, n_jobs=-1, **params).fit(X_train, train_targets)
    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`.

```
In [78]: test_params(max_depth = 5)
```

```
Out[78]: (0.8314262362076011, 0.8370588235294117)
```

```
In [80]: test_params(max_depth = 26)
```

```
Out[80]: (0.9989783408255006, 0.8658823529411764)
```

```
In [81]: test_params(max_leaf_nodes=2**5)
```

```
Out[81]: (0.8400081732733959, 0.84)
```

```
In [82]: test_params(max_leaf_nodes=2**20)
```

```
Out[82]: (1.0, 0.8647058823529412)
```

```
In [83]: test_params(max_depth = 26, max_leaf_nodes = 2**20)
```

```
Out[83]: (0.998467511238251, 0.8658823529411764)
```

```
In [84]: base_accuracies # no max depth or max leaf nodes
```

```
Out[84]: (1.0, 0.8611764705882353)
```

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

## max\_features

Instead of picking all features (columns) for every split, we can specify that only a fraction of features be chosen:

**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. It may seem counterintuitive, choosing all features for every split of every tree will lead to identical trees, so the default is set to a smaller value.

```
In [85]: test_params(max_features = 'log2')
```

```
Out[85]: (1.0, 0.8558823529411764)
```

```
In [86]: test_params(max_features=3)
```

```
Out[86]: (1.0, 0.8564705882352941)
```

```
In [93]: test_params(max_features=10)
```

```
Out[93]: (1.0, 0.8611764705882353)
```

```
In [88]: base_accuracies
```

```
Out[88]: (1.0, 0.8611764705882353)
```

```
In [98]: test_params(min_samples_split=2)
```

```
Out[98]: (1.0, 0.8611764705882353)
```

## 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 in impurity. The default value is 0, and you can increase it to reduce overfitting.

```
In [99]: test_params(min_impurity_decrease = 1e-7)
```

```
Out[99]: (1.0, 0.861764705882353)
```

```
In [106...]: test_params(min_impurity_decrease = 1e-9)
```

```
Out[106...]: (1.0, 0.8629411764705882)
```

## 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, where each tree is trained on a different subset of the data 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 the other trees' data.

```
In [107...]: test_params(bootstrap = False)
```

```
Out[107...]: (1.0, 0.8582352941176471)
```

When bootstrapping is enabled, you can also control the number or fraction of rows to be considered for each tree.

### **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  $(0, 1)$ .

```
In [110...]: test_params(max_samples = .9)
```

```
Out[110...]: (1.0, 0.8635294117647059)
```

Learn more about bootstrapping here: <https://towardsdatascience.com/what-is-out-of-bag-oob-score-in-random-forests-3a2a2a2a2a2a>

## class\_weight

```
In [111...]: model.classes_
```

```
Out[111...]: array(['No', 'Yes'], dtype=object)
```

```
In [112...]: test_params(class_weight='balanced')
```

```
Out[112... (1.0, 0.8570588235294118)
```

```
In [113... test_params(class_weight={'No': 1, 'Yes': 2})
```

```
Out[113... (1.0, 0.8611764705882353)
```

## Strategy for Tuning Hyperparameters

Here's a good strategy for tuning hyperparameters:

1. Tune the most important/impactful hyperparameter first e.g. n\_estimators
2. With the best value of the first hyperparameter, tune the next most impactful hyperparameter
3. And so on, keep training the next most impactful parameters with the best values for previous parameters
4. Then, go back to the top and further tune each parameter again for further marginal gains

Keep your ideas and experiments organized using a google or excel sheet.

Your first objective should be make the training loss as low as possible (even if the validation loss is very large), loss while increasing the training loss.

## Putting it together

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

```
In [119... model = RandomForestClassifier(n_jobs=-1,
                                     random_state=42,
                                     n_estimators=500,
                                     max_features=10,
                                     max_depth=26,
                                     class_weight={'No': 1, 'Yes': 1.5})
```

```
In [120... model.fit(X_train, train_targets)
```

```
Out[120... RandomForestClassifier(class_weight={'No': 1, 'Yes': 1.5}, max_depth=26,
                                 max_features=10, n_estimators=500, n_jobs=-1,
                                 random_state=42)
```

```
In [121... model.score(X_train, train_targets), model.score(X_val, val_targets)
```

```
Out[121... (0.9996935022476502, 0.8623529411764705)
```

```
In [118... base_accuracies
```

```
Out[118... (1.0, 0.8611764705882353)
```

We've increased the accuracy from 80.58% with a single decision tree to 86.23% with a well-tuned random forest. We did not see a significant improvement with hyperparameter tuning.

There are several possible reasons for the observed limitations in our model's performance:

- We might not have identified the optimal combination of hyperparameters to effectively regularize the model.
- The current modeling technique, Random Forests, may have reached its effectiveness limits. Exploring alternative models could be warranted.
- Insufficient data might be a limiting factor. Acquiring more data could potentially enhance the model's predictive power.
- The predictive accuracy may be constrained by the available features. Considering additional features or engineering existing ones could improve performance.
- Predicting whether it will rain tomorrow could be inherently challenging due to the inherent randomness of weather, which sets a fundamental limit on the accuracy achievable with the given data and modeling techniques.

It's important to acknowledge that no model is perfect. If you can rely on the model we've created today to make decisions, that's great, though it may sometimes be wrong.

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

```
In [122]: model.score(X_test, test_targets)
```

```
Out[122]: 0.8340409108452335
```

Notice that the test accuracy is lower

## 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 every time. It's also important to save imputers, scalers, encoders and even column names. Anything that will be required to make predictions must be saved along with the model.

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

## 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
- Making predictions on single inputs

We also introduced the following terms:

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