CAPAIAN PEMBELAJARAN PRAKTIKUM

1. Menjelaskan konsep-konsep dari pembelajaran mesin untuk kemudian bisa diterapkan pada berbagai permasalahan– (C2);
2. Mengimplementasikan algoritma dan/atau metode di pembelajaran mesin sehingga bisa mendapatkan pemecahan masalah dan model yang sesuai – (C3)
3. Menghasilkan suatu model terbaik dari permasalahan dan terus melakukan perbaikan jika terdapat perkembangan di bidang rekayasa cerdas – (C6)

MINGGU 5
Regresi dan Induksi Pohon Keputusan

DESKRIPSI TEMA

Mahasiswa mempelajari algoritma regresi dan pohon keputusan dengan menggunakan Bahasa pemrograman Pyton.

CAPAIAN PEMBELAJARAN MINGGUAN (SUB-CAPAIAN PEMBELAJARAN)

Mahasiswa dapat menerapkan algoritma regresi dan pohon keputusan sehingga bisa menghasilkan model terbaik – SCPMK-05

PERALATAN YANG DIGUNAKAN

Anaconda Python 3
Laptop atau Personal Computer

LANGKAH-LANGKAH PRAKTIKUM

LINEAR REGRESSION

Simple linear regression can be used to model a linear relationship between one response variable and one explanatory variable. Linear regression has been applied to many important scientific and social problem.

In this lecture, we will start with a quick intuitive walk-trough of the mathematics behind the well-known problem, before seeing how before moving on the see how linear models can be generalize to account for more complicated patterns in data.

1. We begin with the standard import :

```
%matplotlib inline
import matplotlib.pyplot as plt
import seaborn as sns; sns.set()
import numpy as np
```

Simple Linear Regression

We will start with the most familiar linear regression, a straight – line fit to data. As straight – line fit is a model of the form :

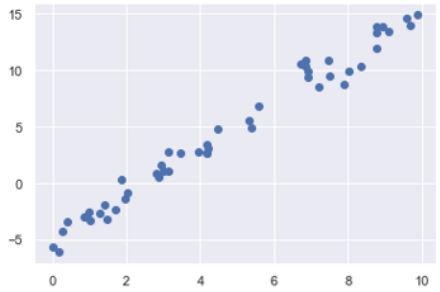
$$y = ax + b$$

where a is a commonly known as the slope and b is commonly known as the intercept.

2. Consider the following data, which is scattered about a line with a slope of 2 and an intercept of -5:

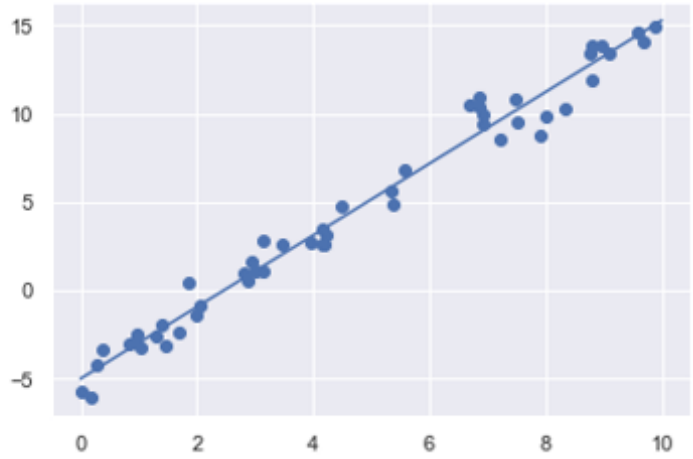
```
rng = np.random.RandomState(1)
x=10*rng.rand(50)
y=2*x-5+rng.randn(50)
plt.scatter(x,y)
```

```
Out[3]: <matplotlib.collections.PathCollection at 0x211b36d9630>
```



3. We can use Scikit-Learn's LinearRegression estimator to fit this data and construct the best-fit line :

```
:
1  from sklearn.linear_model import LinearRegression
2  model = LinearRegression(fit_intercept=True)
3
4  model.fit(x[:, np.newaxis], y)
5
6  xfit = np.linspace(0, 10, 1000)
7  yfit = model.predict(xfit[:, np.newaxis])
8
9  plt.scatter(x, y)
10 plt.plot(xfit, yfit);
```



4. The slope and intercept of the data are contained in the model's fit parameters, which in Scikit Learn are always marked by a trailing underscore. Here the relevant parameters are coef_ and intercept_:

```
1  print("Model slope:    ", model.coef_[0])
2  print("Model intercept:", model.intercept_)
```

```
Model slope:     2.027208810360695
Model intercept: -4.998577085553202
```

We see that the result are very close to the inputs, as we might hope.
The LinearRegression estimator is much more capable than this, however-in additional to simple straight – line fits, it can also handle multidimensional linear models.

5. The multidimensional nature such as regression makes them more difficult to visualize, but we can see one pf these fits in action by building some example data, using Numpy's matrix multiplication operator :

```
rng = np.random.RandomState(1)
X = 10 * rng.rand(100, 3)
y = 0.5 + np.dot(X, [1.5, -2., 1.])

model.fit(X, y)
print(model.intercept_)
print(model.coef_)
```

```
0.5000000000000144
[ 1.5 -2.   1. ]
```

Here the y data is constructed from three random x values and the linear regression recovers the coefficients used to construct the data.
In this way, we can use single LinearRegression estimator to fir lines, planes, or hyperplanes to our data. It still appears that this approach would be limited but strictly linear relationship between variables.

BASIS FUNCTION REGRESSION
One trick we can use to adapt linear regression to non linear relationships between variables is to transform the data according to basis functions. The idea is to take our multidimensional linear model :

$$y = a_0 + a_1x_1 + a_2x_2 + a_2x_2 + \cdots$$

And build the x_1, x_2, x_3 and so on from our single – dimensional input x. That is we let $x_n = f_n(x)$ where $f_n(\,)$ is some function that transform our data.
For example, $f_n(x) = x^n$, our model becomes a polynomial regression :

$$y = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots$$

Notice, that still a linear model – the linearity refers to the fact that the coefficient a_n never multiply or divide each other. What we have effectively done is taken our one-dimensional x values and projected them into a higher dimension, so that a linear fit can fit more complicated relationship between x and y.

Polynomial Basis Functions

6. This polynomial projection is useful enough that is built into Scikit-Learn, using the PolynomialFeatures transformer :

```
from sklearn.preprocessing import PolynomialFeatures
x = np.array([2, 3, 4])
poly = PolynomialFeatures(3, include_bias=False)
poly.fit_transform(x[:, None])
```

```
array([[ 2.,  4.,  8.],
       [ 3.,  9., 27.],
       [ 4., 16., 64.]])
```

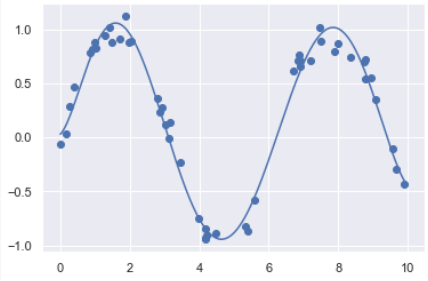
We see here that the transformer has converted our dimensional array into a three dimensional array by taking the exponent of each value. This new, higher dimensional data representation can then be plugged into a linear regression. The cleanest way to accomplish this is to use a pipeline.

7. Lets make a 7-th degree polynomial model this way :

```
from sklearn.pipeline import make_pipeline
poly_model = make_pipeline(PolynomialFeatures(7),
                           LinearRegression())
```

8. With this transform in place, we can use the linear model to fit more complicated relationship between x and y. For example, her is a sine wave with noise :

```
rng = np.random.RandomState(1)
x = 10 * rng.rand(50)
y = np.sin(x) + 0.1 * rng.randn(50)

poly_model.fit(x[:, np.newaxis], y)
yfit = poly_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit);
```



Our linear model, through the use of 7th –order polynomial basis function, can provide an excellent fit to this nonlinear data.

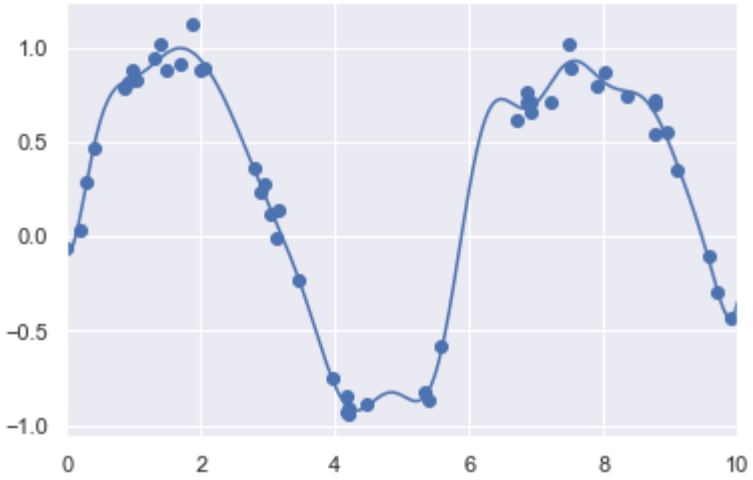
Gaussian Basis Function

Of course, other basis function functions are possible. For example, one useful pattern is to fit a model that is not a sum of polynomial bases, but a sum of Gaussian bases. These Gaussian basis functions are not build into Scikit-Learn, but we can write a custom transformer.

9. Scikit-learn transformers are implemented as Python classes; reading Scikit-Learn source is a good way to see how they can be created :

```
from sklearn.base import BaseEstimator, TransformerMixin

class GaussianFeatures(BaseEstimator, TransformerMixin):
    """Uniformly spaced Gaussian features for one-dimensional input"""

    def __init__(self, N, width_factor=2.0):
        self.N = N
        self.width_factor = width_factor

    @staticmethod
    def _gauss_basis(x, y, width, axis=None):
        arg = (x - y) / width
        return np.exp(-0.5 * np.sum(arg ** 2, axis))

    def fit(self, X, y=None):
        # create N centers spread along the data range
        self.centers_ = np.linspace(X.min(), X.max(), self.N)
        self.width_ = self.width_factor * (self.centers_[1] - self.centers_[0])
        return self

    def transform(self, X):
        return self._gauss_basis(X[:, :, np.newaxis], self.centers_,
                                 self.width_, axis=1)

gauss_model = make_pipeline(GaussianFeatures(20),
                            LinearRegression())
gauss_model.fit(x[:, np.newaxis], y)
yfit = gauss_model.predict(xfit[:, np.newaxis])

plt.scatter(x, y)
plt.plot(xfit, yfit)
plt.xlim(0, 10);

```

So, the figure a Gaussian basis function fit computed with a custom transformer, it's shown below :



We put this example here just to make clear that there is nothing 'magic' about polynomial basis function :if we have some sort of intuition into the generating process off our data that makes we think one basis or another might be appropriate, we can use them as well.

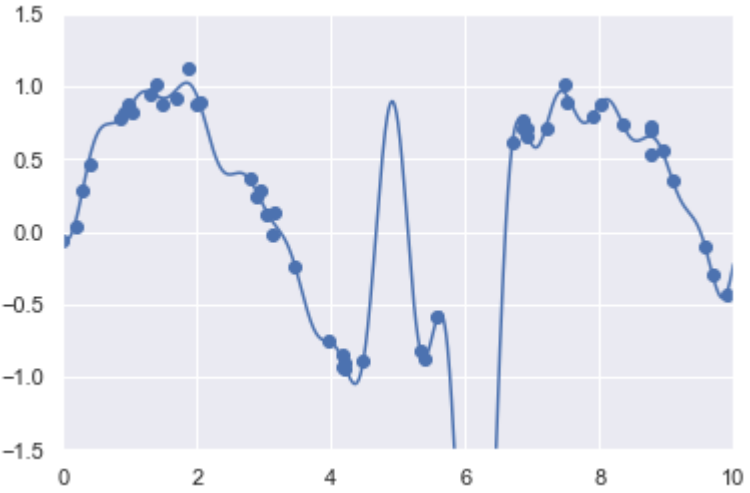
Regularization

The introduction of basis function into our linear regression makes the model much more flexible, but it also can very quickly lead to overfitting. For example : if we choose too many Gaussian basis functions, we end up with result that don't look so good.

10. Let's observe this code below :

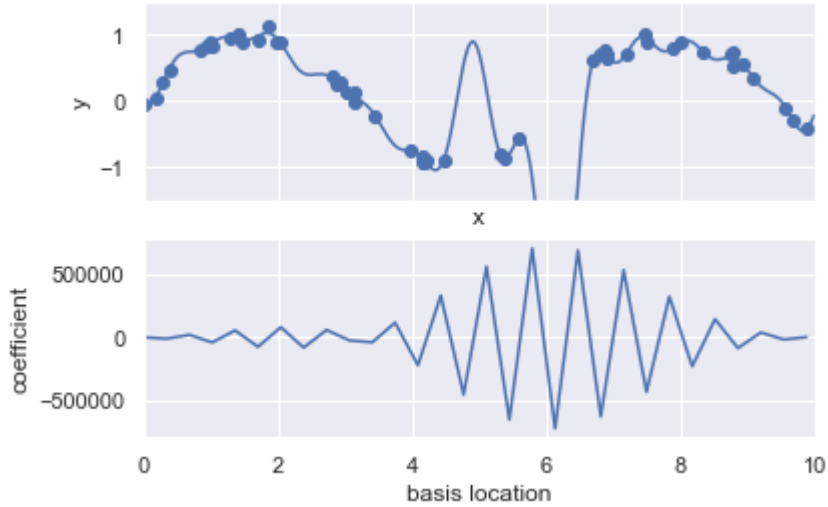
```
model = make_pipeline(GaussianFeatures(30),
                      LinearRegression())
model.fit(x[:, np.newaxis], y)

plt.scatter(x, y)
plt.plot(xfit, model.predict(xfit[:, np.newaxis]))

plt.xlim(0, 10)
plt.ylim(-1.5, 1.5);
```

The output, shown us an overly complex basis function model that overfits the data.



With the data projected to the 30-dimensional basis, the model has far too much flexibility and goes to extreme values between locations where it is constrained by data. We can see the reason for this if we plot the coefficients of the Gaussian bases with respect to their locations :

```
def basis_plot(model, title=None):
    fig, ax = plt.subplots(2, sharex=True)
    model.fit(x[:, np.newaxis], y)
    ax[0].scatter(x, y)
    ax[0].plot(xfit, model.predict(xfit[:, np.newaxis]))
    ax[0].set(xlabel='x', ylabel='y', ylim=(-1.5, 1.5))

    if title:
        ax[0].set_title(title)

    ax[1].plot(model.steps[0][1].centers_,
               model.steps[1][1].coef_)
    ax[1].set(xlabel='basis location',
              ylabel='coefficient',
              xlim=(0, 10))

model = make_pipeline(GaussianFeatures(30), LinearRegression())
basis_plot(model)
```



The figure above, shown us the coefficients of the Gaussian bases in the overly complex model. The lower panel shows the amplitude of the basis functions at each location. This is typical overfitting behavior when basis function overlap : the coefficients of adjacent basis functions blow p and cancel each other out. We know that such behavior is problematic, and it would be nice if we could limit such spikes explicitly in the model by penalizing large values of the model parameters. Such a penalty is known as *regularization*, and comes in several forms.

Ridge Regression (L_2 regularization)

Perhaps the most common form of regularization is known as ridge regression, sometimes also called Tikhonov regularization. This proceeds by penalizing the sum of squares (2-norms) of the model coefficients : in this case, the penalty on the model fit would be :

$$P = \alpha \sum_{n=1}^{N} \theta_n^2$$

Where α is a free parameter that controls the strength of the penalty.

11. Ridge regression is a type of penalized model. If we build using Scikit-Learn, we use *Ridge* estimator

```
from sklearn.linear_model import Ridge
model = make_pipeline(GaussianFeatures(30), Ridge(alpha=0.1))
basis_plot(model, title='Ridge Regression')
```

The parameter α is essentially a knob controlling the complexity of the resulting model. In the limit $\alpha \rightarrow 0$, we recover the standard linear regression result, in the limit $\alpha \rightarrow \infty$, all model responses will be suppressed. One advantage of ridge regression in particular is that it can be computed very efficiently – at hardly more computational cost than the original linear regression model.

Lasso Regression (L_1 regularization)

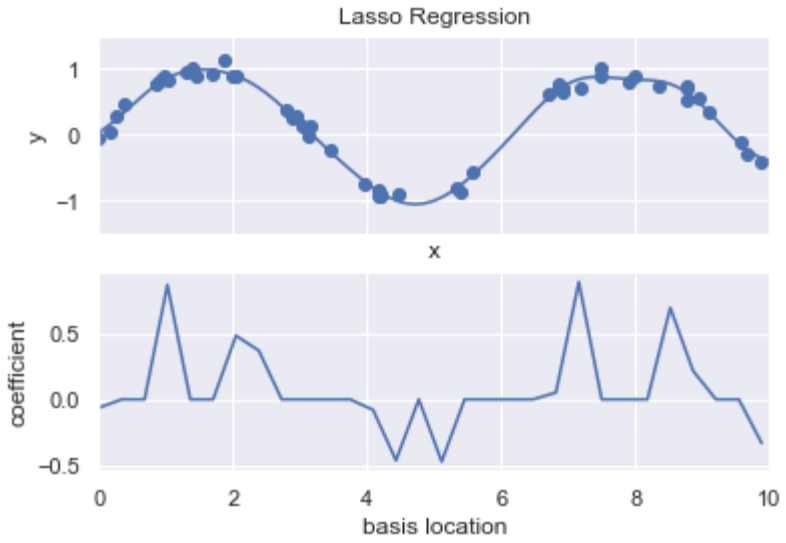
Another common type of regularization, and involves penalizing the sum of absolute values (1-norms) of regression coefficients :

$$P = \alpha \sum_{n=1}^{N} |\theta_n|$$

Though this is conceptually very similar to ridge regression, the results can differ surprisingly: for example, due to geometric reasons lasso regression tends to favor *sparse models* where possible: that is, it preferentially sets model coefficients to exactly zero.

12. We can see this behavior in duplicating the ridge regression figure, but using L1-normalized coefficients :

```
from sklearn.linear_model import Lasso
model = make_pipeline(GaussianFeatures(30), Lasso(alpha=0.001))
basis_plot(model, title='Lasso Regression')
```



With the lasso regression penalty, the majority of the coefficients are exactly zero, with the functional behavior being modeled by a small subset of the available basis functions. As with ridge regularization, the parameter α tunes the strength of the penalty, and should be determined via, for example, cross-validation.

Linear Regression Example : Predicting Bicycle Traffic

As an example, let's take a look at whether we can predict the number of bicycle trips across Seattle's Freemont Bridge based on weather, season and other factors.

In this section, we will join the bike data with another dataset, and try to determine the extent to which weather and seasonal factors – temperature, precipitation, and daylight hours – affect the volume of bicycle traffic through this corridor. We will perform a simple linear regression to relate weather and other information to bicycle counts, in order to estimate how a change in any one of these parameters affects the number of riders on given day.

13. Import the dataset (provide in your e-learning) :

```
import pandas as pd
counts = pd.read_csv('FremontBridge.csv', index_col='Date', parse_dates=True)
weather = pd.read_csv('BicycleWeather.csv', index_col='DATE', parse_dates=True)
```

14. Next we will compute the total daily bicycle traffic, and put this in this own dataframe :

```
daily = counts.resample('d').sum()
daily['Total'] = daily.sum(axis=1)
daily = daily[['Total']] # remove other columns
```

15. We saw previously that the patterns of use generally vary from day to day; let's account for this in our data by adding binary columns that indicate the day of the week :

```
days = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun']
for i in range(7):
    daily[days[i]] = (daily.index.dayofweek == i).astype(float)
```

16. Similarly, we might expect riders to behave differently on holidays; let's add indicator of this as well :

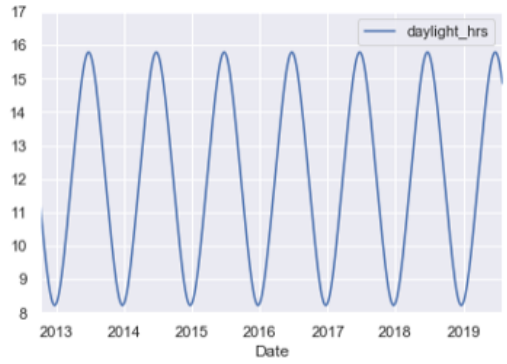
```
from pandas.tseries.holiday import USFederalHolidayCalendar
cal = USFederalHolidayCalendar()
holidays = cal.holidays('2012', '2016')
daily = daily.join(pd.Series(1, index=holidays, name='holiday'))
daily['holiday'].fillna(0, inplace=True)
```

17. We also might suspect that the hours of daylight would affect how many people ride; let's use the standard calculation to add this information :

```
def hours_of_daylight(date, axis=23.44, latitude=47.61):
    """Compute the hours of daylight for the given date"""
    days = (date - pd.datetime(2000, 12, 21)).days
    m = (1. - np.tan(np.radians(latitude))
         * np.tan(np.radians(axis) * np.cos(days * 2 * np.pi / 365.25)))
    return 24. * np.degrees(np.arccos(1 - np.clip(m, 0, 2))) / 180.

daily['daylight_hrs'] = list(map(hours_of_daylight, daily.index))
daily[['daylight_hrs']].plot()
plt.ylim(8, 17)
```

Here's, the visualization of hours of daylight in Seattle :

```
(8, 17)
```



18. We can also add the average temperature and total precipitation to the data. In addition to the inches of precipitation; let's add a flag that indicates whether a day is dry (has zero precipitation)

```
# temperatures are in 1/10 deg C; convert to C
weather['TMIN'] /= 10
weather['TMAX'] /= 10
weather['Temp (C)'] = 0.5 * (weather['TMIN'] + weather['TMAX'])

# precip is in 1/10 mm; convert to inches
weather['PRCP'] /= 254
weather['dry day'] = (weather['PRCP'] == 0).astype(int)

daily = daily.join(weather[['PRCP', 'Temp (C)', 'dry day']])
```

19. Finally, let's add a counter that increases from day 1, and measures how many years have passes. This will let us measure any observed annual increase of decrease in daily crossing :

```
daily['annual'] = (daily.index - daily.index[0]).days / 365.
```

20. Now our data is in order, and we can take a look at it :

```
daily.head()
```

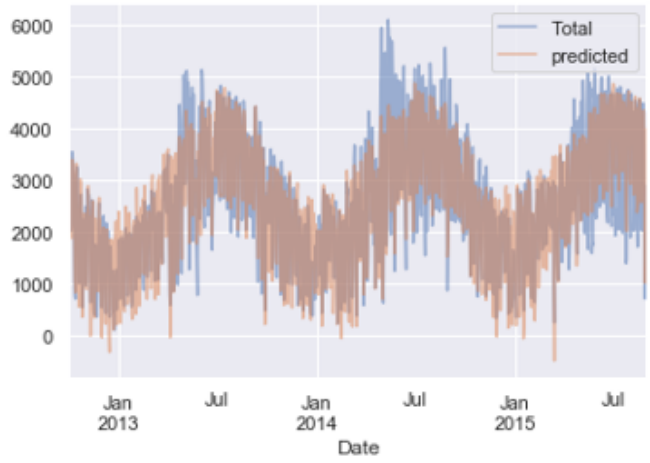
Date	Total	Mon	Tue	Wed	Thu	Fri	Sat	Sun	holiday	daylight_hrs	PRCP	Temp (C)	dry day	annual
2012-10-03	3521.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	0.0	11.277359	0.0	13.35	1.0	0.000000
2012-10-04	3475.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	0.0	11.219142	0.0	13.60	1.0	0.002740
2012-10-05	3148.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	0.0	11.161038	0.0	15.30	1.0	0.005479
2012-10-06	2006.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	0.0	11.103056	0.0	15.85	1.0	0.008219
2012-10-07	2142.0	0.0	0.0	0.0	0.0	0.0	0.0	1.0	0.0	11.045208	0.0	15.85	1.0	0.010959

21. With this in place, we can choose the columns to use and fit a linear regression model to our data. We will set fit_intercepts =False, because the daily flags essentially operate as their own day-specific intercepts :

```
# Drop any rows with null values
daily.dropna(axis=0, how='any', inplace=True)

column_names = ['Mon', 'Tue', 'Wed', 'Thu', 'Fri', 'Sat', 'Sun', 'holiday',
                'daylight_hrs', 'PRCP', 'dry day', 'Temp (C)', 'annual']
X = daily[column_names]
y = daily['Total']

model = LinearRegression(fit_intercept=False)
model.fit(X, y)
daily['predicted'] = model.predict(X)
```

22. Finally, we can compare the total and predicted bicycle traffic visually, and the visualization of our model's prediction of bicycle traffic :

```
daily[['Total', 'predicted']].plot(alpha=0.5);
```



In this evident we have missed some key features, especially during the summer time. Either our features are not complete (i.e., people decide whether to ride to work based on more than just these) or there are some nonlinear relationships that we have failed to take into account (e.g., perhaps people ride less at both high and low temperatures).

23. Nevertheless, our rough approximation is enough to give us some insights, and we can take a look at the coefficients of the linear model to estimate how much each feature contributes to the daily bicycle count :

```
params = pd.Series(model.coef_, index=X.columns)
params
```

```
Mon             504.882756
Tue             610.233936
Wed             592.673642
Thu             482.358115
Fri             177.980345
Sat           -1103.301710
Sun           -1133.567246
holiday       -1187.401381
daylight_hrs    128.851511
PRCP           -664.834882
dry day         547.698592
Temp (C)         65.162791
annual           26.942713
dtype: float64
```

24. These number are difficult to interpret without some measure of their uncertainty. We can compute these uncertainties quickly using bootstrap resampling of the data :

```
from sklearn.utils import resample
np.random.seed(1)
err = np.std([model.fit(*resample(X, y)).coef_
              for i in range(1000)], 0)
```

25. With these errors estimated, let's again look at the results :

```
print(pd.DataFrame({'effect': params.round(0),
                    'error': err.round(0)}))
```

```
              effect  error
Mon            505.0   86.0
Tue            610.0   83.0
Wed            593.0   83.0
Thu            482.0   85.0
Fri            178.0   81.0
Sat          -1103.0   80.0
Sun          -1134.0   83.0
holiday      -1187.0  163.0
daylight_hrs   129.0    9.0
PRCP          -665.0   62.0
dry day        548.0   33.0
Temp (C)        65.0    4.0
annual          27.0   18.0
```

We first see that there is relatively stable trend in the weekly baseline : there are many more riders on weekdays that on weekends and holidays. We see that for each additional hour of daylight, 129 ± 9 more people choose to ride; a temperature increase of one degree Celsius encourages 65 ± 4 people to grab their bicycle; a dry means an average of 548 ± 33 more riders, and each inch precipitation means 665 ± 62 more people leave their bike at home. Once all these effects are accounted for, we see a modest increase of 27 ± 18 new daily rides each riders each year.

Our model is almost certainly missing some relevant information. For example, nonlinear effects (such as effects of precipitation *and* cold temperature) and nonlinear trends within each variable (such as disinclination to ride at very cold and very hot temperatures) cannot be accounted for in this model. Additionally, we have thrown away some of the finer-grained information (such as the difference between a rainy morning and a rainy afternoon), and we have ignored correlations between days (such

as the possible effect of a rainy Tuesday on Wednesday's numbers, or the effect of an unexpected sunny day after a streak of rainy days).

DECISION TREES LEARNING

Decision trees are widely used models for classification and regression tasks. Essentially, they learn a hierarchy of if/else questions, leading to a decision.

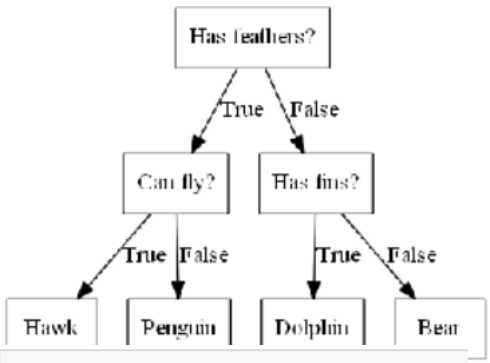
1. Download the graphviz library (.zip) from https://graphviz.gitlab.io/_pages/Download/Download_windows.html
1. Unzip and place it to your system path (commonly on Program Files (x86)).
2. Install graphviz package from jupyter notebook.

```
!pip install mglearn
```

```
Requirement already satisfied: mglearn in d:\umn\lecturers\apps\lib\site-packages (0.1.7)
Requirement already satisfied: numpy in d:\umn\lecturers\apps\lib\site-packages (from mglearn) (1.15.4)
Requirement already satisfied: scikit-learn in d:\umn\lecturers\apps\lib\site-packages (from mglearn) (0.20.1)
Requirement already satisfied: matplotlib in d:\umn\lecturers\apps\lib\site-packages (from mglearn) (3.0.2)
Requirement already satisfied: imageio in d:\umn\lecturers\apps\lib\site-packages (from mglearn) (2.4.1)
Requirement already satisfied: pandas in d:\umn\lecturers\apps\lib\site-packages (from mglearn) (0.23.4)
Requirement already satisfied: pillow in d:\umn\lecturers\apps\lib\site-packages (from mglearn) (5.3.0)
Requirement already satisfied: cycler in d:\umn\lecturers\apps\lib\site-packages (from mglearn) (0.10.0)
Requirement already satisfied: scipy>=0.13.3 in d:\umn\lecturers\apps\lib\site-packages (from scikit-learn->mglearn) (1.1.0)
Requirement already satisfied: kiwisolver>=1.0.1 in d:\umn\lecturers\apps\lib\site-packages (from matplotlib->mglearn) (1.0.1)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in d:\umn\lecturers\apps\lib\site-packages (from matplo
tlib->mglearn) (2.3.0)
Requirement already satisfied: python-dateutil>=2.1 in d:\umn\lecturers\apps\lib\site-packages (from matplotlib->mglearn) (2.7.
5)
Requirement already satisfied: pytz>=2011k in d:\umn\lecturers\apps\lib\site-packages (from pandas->mglearn) (2018.7)
Requirement already satisfied: six in d:\umn\lecturers\apps\lib\site-packages (from cycler->mglearn) (1.12.0)
Requirement already satisfied: setuptools in d:\umn\lecturers\apps\lib\site-packages (from kiwisolver>=1.0.1->matplotlib->mglea
rn) (40.6.3)
```

3. Install mglearn dataset from jupyter notebook.
4. Import libraries :
5. Build decision tree from animal dataset.

```
mglearn.plots.plot_animal_tree()
```

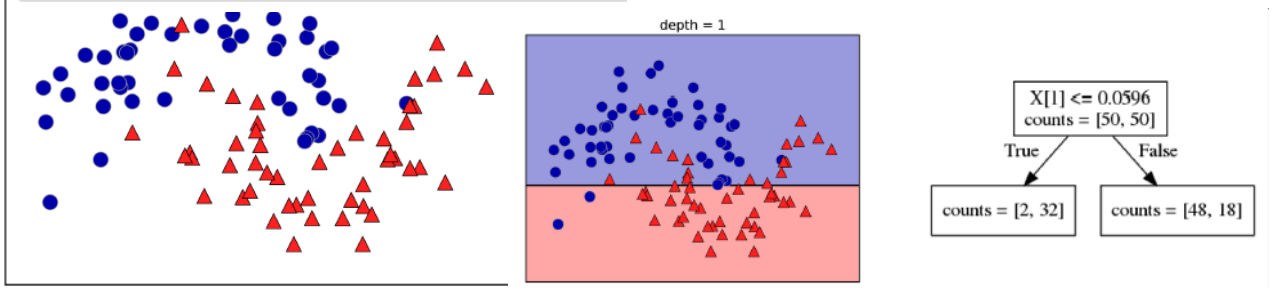



```
import numpy as np
import sklearn.datasets
import mglearn
import matplotlib.pyplot as plt
```

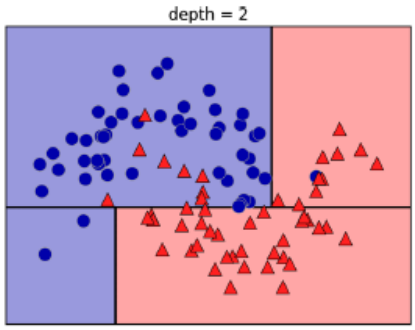
Building Decision Trees

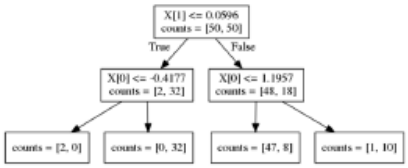
6. Let's go through the process of building a decision tree for the 2D classification dataset. Dataset: two_moons, consists of two half-moon shapes, each consisting of 50 data points.

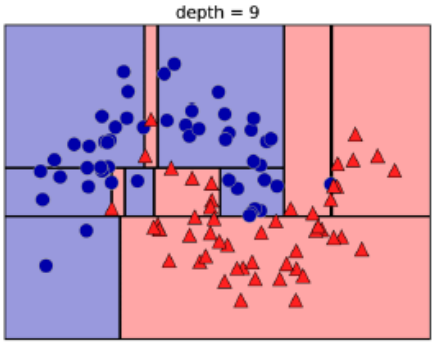
```
1  mglearn.plots.plot_tree_progressive()
```

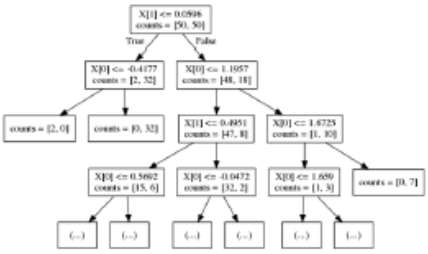


Splitting the dataset vertically at x[1]=0.0596 yields the most information; it best separates the points in class 1 from the points in class 2. The split is done by testing whether x[1] <= 0.0596, indicated









by a black line.We can build a more accurate model by repeating the process of looking for the best test in both regions. the most informative next split for the left and the right region is based on x[0].

Controlling Complexity of Decision Trees

Typically, building a tree as described here and continuing until all leaves are pure leads to models that are very complex and highly overfit to the training data. There are two common strategies to prevent overfitting:

1. stopping the creation of the tree early (also called *pre-pruning*)
2. building the tree but then removing or collapsing nodes that contain little information (also called *post-pruning* or just *pruning*)

7. Decision trees in scikit-learn are implemented in the DecisionTreeRegressor and DecisionTreeClassifier classes. Scikit-learn only implements pre-pruning, not post-pruning.

```
from sklearn.tree import DecisionTreeClassifier
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split

cancer = load_breast_cancer()
x_train, x_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, stratify=cancer.target, random_state=42)
tree = DecisionTreeClassifier(random_state=0)
tree.fit(x_train, y_train)
print("Accuracy on training set: {:.3f}".format(tree.score(x_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(x_test, y_test)))
```

```
Accuracy on training set: 1.000
Accuracy on test set: 0.937
```

As expected, the accuracy on the training set is 100%—because the leaves are pure, the tree was grown deep enough that it could perfectly memorize all the labels on the training data. The test set accuracy is slightly worse than for the linear models we looked at previously, which had around 95% accuracy.

8. Let's apply pre-pruning to the tree, which will stop developing the tree before we perfectly fit to the training data. Set max_depth=4, meaning only four consecutive questions can be asked.

```
tree = DecisionTreeClassifier(max_depth=4, random_state=0)
tree.fit(x_train, y_train)

print("Accuracy on training set: {:.3f}".format(tree.score(x_train, y_train)))
print("Accuracy on test set: {:.3f}".format(tree.score(x_test, y_test)))
```

```
Accuracy on training set: 0.988
Accuracy on test set: 0.951
```

Analying Decision Trees

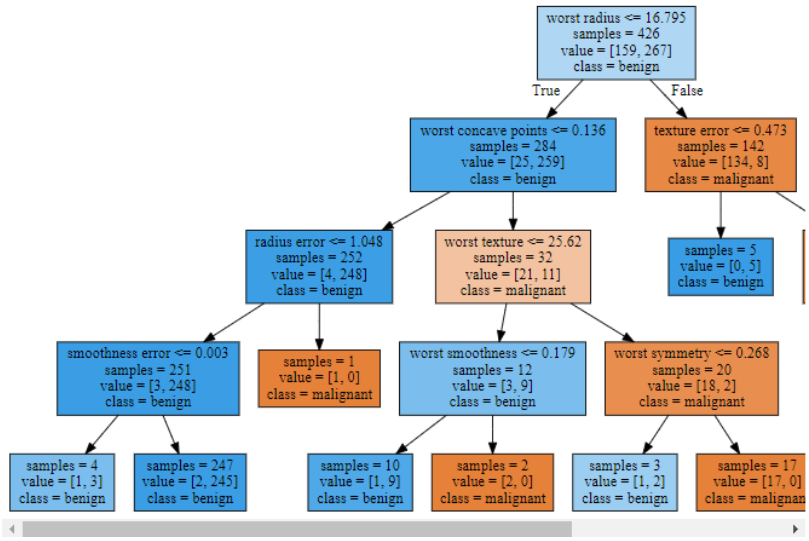
9. Visualize the tree using the export_graphviz function from the tree module.

It writes a file in the .dot file format, which is a text file format for storing graphs.

```
from sklearn.tree import export_graphviz
export_graphviz(tree, out_file="tree.dot", class_names=["malignant", "benign"],
                feature_names=cancer.feature_names, impurity=False, filled=True)
```

```
import graphviz

with open("tree.dot") as f:
    dot_graph = f.read()
graphviz.Source(dot_graph)
```

Out[13]:



The n_samples shown in each node gives the number of samples in that node. Value provides the number of samples per class.

Feature Importance in Trees

There are some useful properties that we can derive to summarize the workings of the tree. One of them is feature importance, which rates how important each feature is for the decision a tree makes. The feature importances always sum to 1.

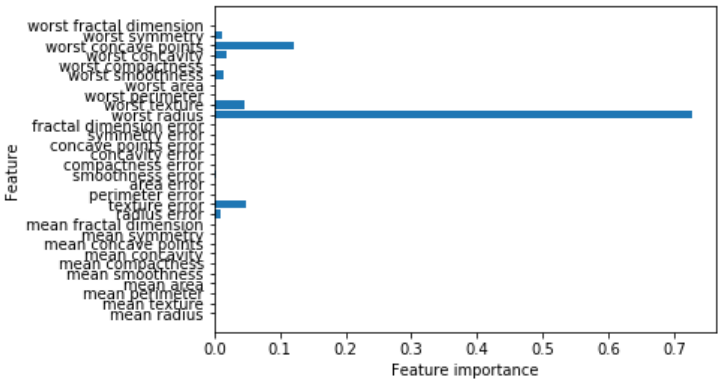
10. Show the feature importances.

```
print("Feature importances:\n{}".format(tree.feature_importances_))
```

```
Feature importances:
[0.         0.         0.         0.         0.         0.
 0.         0.         0.         0.         0.01019737 0.04839825
 0.         0.         0.0024156  0.         0.         0.
 0.         0.         0.72682851 0.0458159  0.         0.
 0.0141577  0.         0.018188   0.1221132  0.01188548 0.        ]
```

11. Visualize the feature importances.

```
def plot_feature_importances_cancer(model):
    n_features = cancer.data.shape[1]
    plt.barh(range(n_features), model.feature_importances_, align='center')
    plt.yticks(np.arange(n_features), cancer.feature_names)
    plt.xlabel("Feature importance")
    plt.ylabel("Feature")

plot_feature_importances_cancer(tree)
```

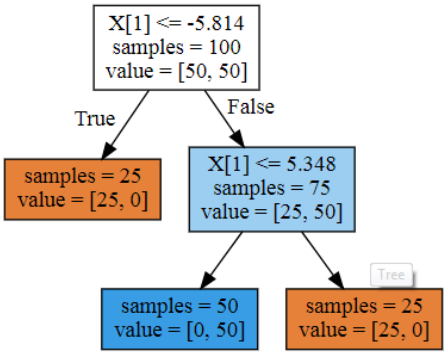


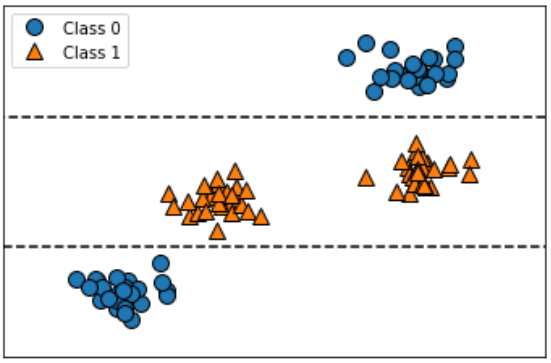
If a feature has a low feature_importance, it doesn't mean that this feature is uninformative. It only means that the feature was not picked by the tree, likely because another feature encodes the same information.

12. In contrast to the coefficients in linear models, feature importances are always positive, and don't encode which class a feature is indicative of.

```
tree = mglearn.plots.plot_tree_not_monotone()
display(tree)
```

```
Feature importances: [0. 1.]
```



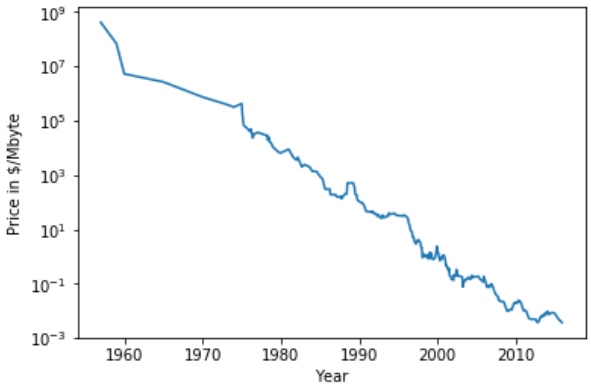


The plot shows a dataset with two features and two classes. Here, all the information is contained in X[1], and X[0] is not used at all. But the relation between X[1] and the output class is not monotonous, meaning we cannot say "a high value of X[0] means class 0, and a low value means class 1" (or vice versa).

13. Let's use a dataset of historical computer memory with the date on the x-axis and the price of one megabyte of RAM in that year on the y-axis(RAM) prices.

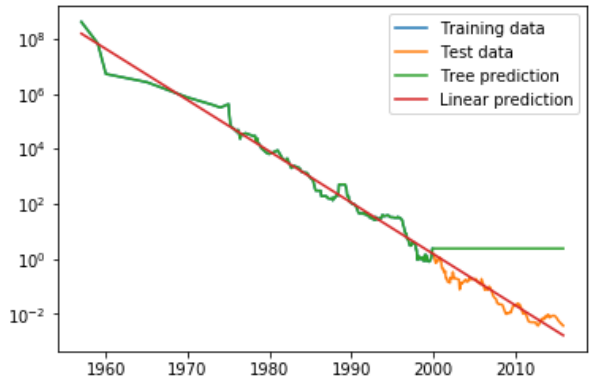
```
1  ram_prices = pd.read_csv(os.path.join(mglearn.datasets.DATA_PATH, "ram_price.csv"))
2
```

```
Out[20]: Text(0,0.5,'Price in $/Mbyte')
```



14. Make a forecast for the years after 2000 using the historical data up to that point, with the date as our only feature. We will compare two simple models: a DecisionTreeRegressor and LinearRegression.

```
Out[22]: <matplotlib.legend.Legend at 0xc4238d0>
```



The tree has no ability to generate "new" responses, outside of what was seen in the training data. This shortcoming applies to all models based on trees.

Random Forests

A random forest is essentially a collection of decision trees, where each tree is slightly different from the others. To implement this strategy, we need to build many decision trees. Each tree should do an acceptable job of predicting the target, and should also be different from the other trees.

2. Analyzing Random Forests

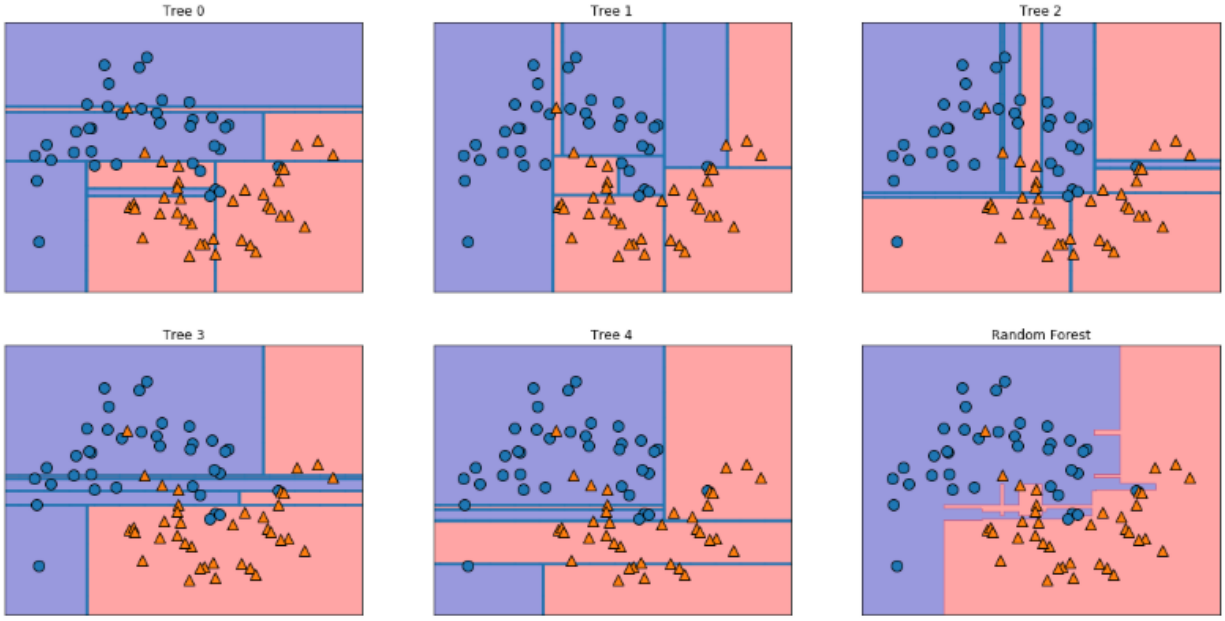
```
from sklearn.tree import DecisionTreeRegressor
from sklearn.linear_model import LinearRegression

# use historical data to forecast prices after the year 2000
data_train = ram_prices[ram_prices.date < 2000]
data_test = ram_prices[ram_prices.date >= 2000]

# predict prices based on date
x_train = data_train.date[:, np.newaxis]
# we use a log-transform to get a simpler relationship of data to target
y_train = np.log(data_train.price)

tree = DecisionTreeRegressor().fit(x_train, y_train)
linear_reg = LinearRegression().fit(x_train, y_train)

# predict on all data
x_all = ram_prices.date[:, np.newaxis]

pred_tree = tree.predict(x_all)
pred_lr = linear_reg.predict(x_all)

# undo log-transform
price_tree = np.exp(pred_tree)
price_lr = np.exp(pred_lr)
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.datasets import make_moons

x, y = make_moons(n_samples=100, noise=0.25, random_state=3)
x_train, x_test, y_train, y_test = train_test_split(x, y, stratify=y, random_state=42)

forest = RandomForestClassifier(n_estimators=5, random_state=2)
forest.fit(x_train, y_train)
```

```
Out[23]: RandomForestClassifier(bootstrap=True, class_weight=None, criterion='gini',
                                max_depth=None, max_features='auto', max_leaf_nodes=None,
                                min_impurity_decrease=0.0, min_impurity_split=None,
                                min_samples_leaf=1, min_samples_split=2,
                                min_weight_fraction_leaf=0.0, n_estimators=5, n_jobs=1,
                                oob_score=False, random_state=2, verbose=0, warm_start=False)
```

15. The trees that are built as part of the random forest are stored in the estimator_ attribute.
16. Let's visualize the decision boundaries learned by each tree, together with their aggregate prediction as made by the forest.

```
fig, axes = plt.subplots(2, 3, figsize=(20,10))
for i, (ax, tree) in enumerate(zip(axes.ravel(), forest.estimators_)):
    ax.set_title("Tree {}".format(i))
    mglearn.plots.plot_tree_partition(x_train, y_train, tree, ax=ax)

mglearn.plots.plot_2d_separator(forest, x_train, fill=True, ax=axes[-1, -1], alpha=.4)
axes[-1, -1].set_title("Random Forest")
mglearn.discrete_scatter(x_train[:, 0], x_train[:, 1], y_train)
```

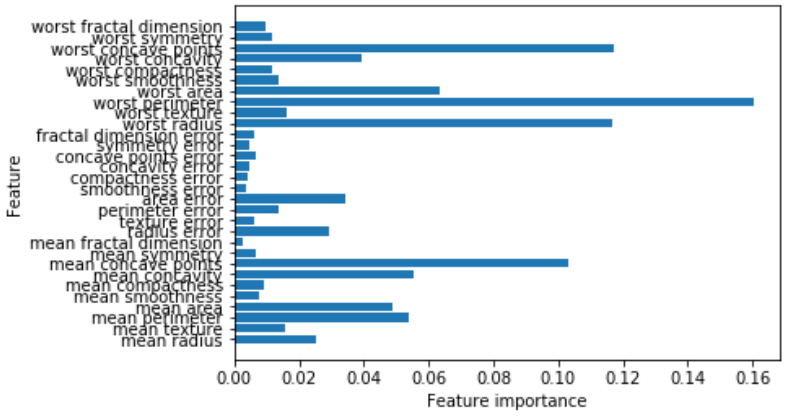


The random forest overfits less than any of the trees individually, and provides a much more intuitive decision boundary.

17. With dataset Breast Cancer.

```
x_train, x_test, y_train, y_test = train_test_split(
    cancer.data, cancer.target, random_state=0)
forest = RandomForestClassifier(n_estimators=100, random_state=0)
forest.fit(x_train, y_train)

print("Accuracy on training set: {:.3f}".format(forest.score(x_train, y_train)))
print("Accuracy on test set: {:.3f}".format(forest.score(x_test, y_test)))
```

```
Accuracy on training set: 1.000
Accuracy on test set: 0.972
```



REFERENSI

1. Van derPlas J. 2016. Pyton Data Science Handbook. Oreilly Media Inc