Machine Learning with Scikit-learn

Part One: Linear and Logistic Regression

Michael Galarnyk, LinkedIn Learning

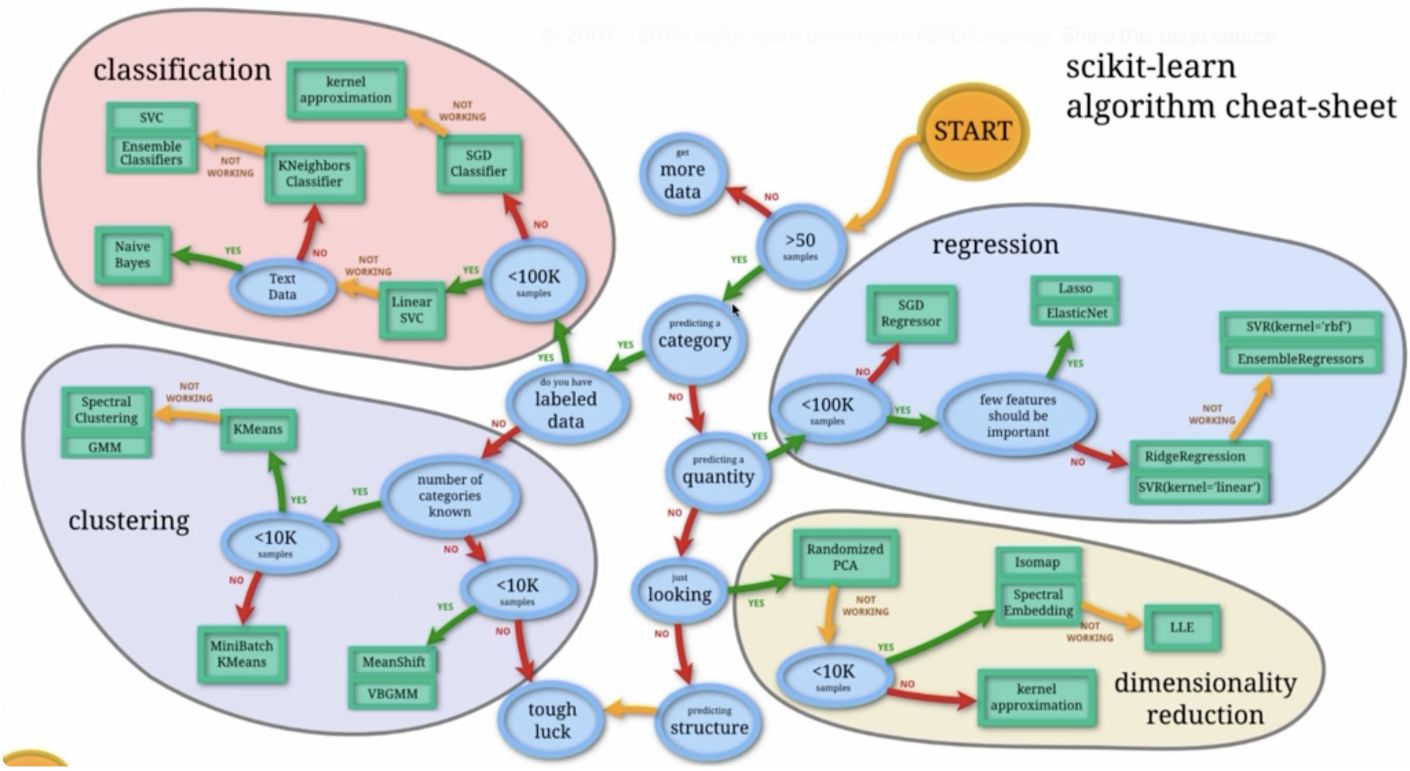
Scikit-learn is a great library for creating machine learning models from data. Before you can fit a model using scikit-learn, your data has to be in a recognizable format. Scikit-learn works well with numeric data that is stored in numpy arrays. Additionally, you can convert your data from objects like pandas dataframes to numpy arrays. In this video, I'll show you how you make your data a more acceptable input for scikit-learn.

⇨ Choosing a Model:

Here are a few things to consider when choosing a model.

- The first thing is a problem you're trying to solve. For example, if you have a supervised learning problem, figuring out if you're trying to predict a continuous or categorical value can be an important first step.
- Next, always consider the size, quality, and structure of your data. There's no machine learning without data.
- You should also consider the strengths and weaknesses of each algorithm you're considering. It's especially important, as some algorithms take longer to make predictions.
- Also, more complex models are often more difficult to maintain. Finally, consider the urgency of a task. Some models take longer to train and tune.

→ Scikit-Learn Model Choosing Cheat Sheet:

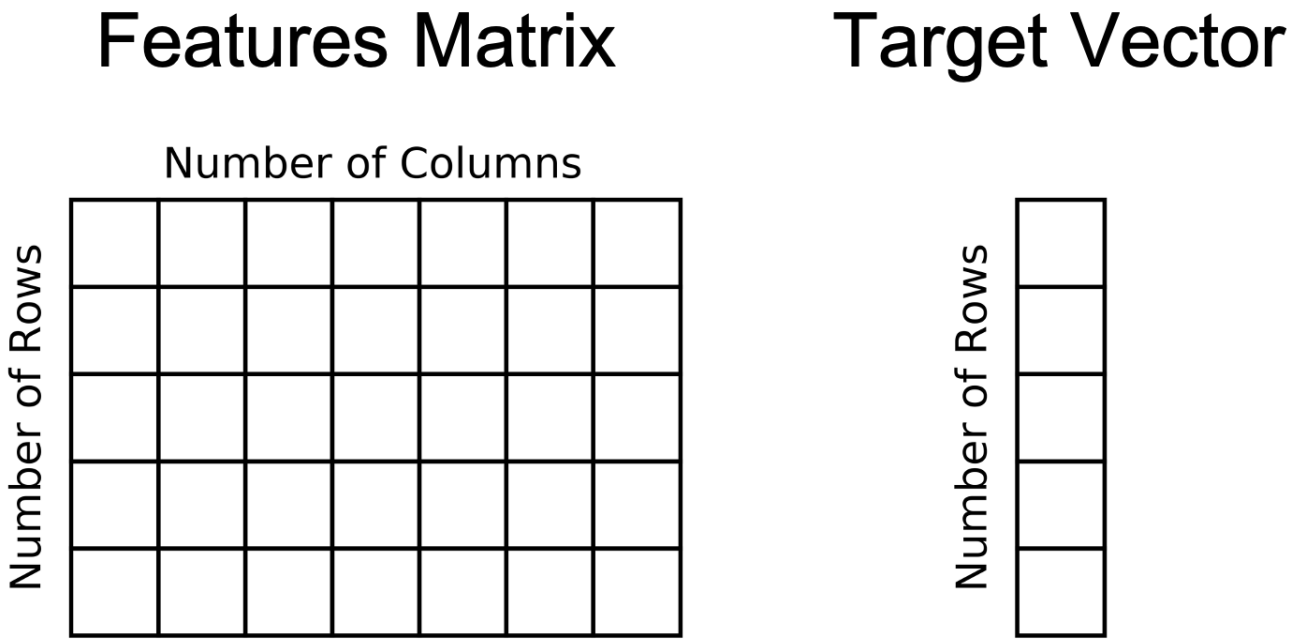


⇨ Supervised Learning

The most common form of machine learning is supervised learning. In Scikit-Learn, a supervised learning algorithm learns a relationship between your features matrix and your target factor. A feature is a measurable property. A target is typically what you want to make predictions for. Once a model learns a relationship between a features matrix and a target factor, it can make predictions for unseen or future data. Supervised learning can generally be thought of to solve two different types of tasks. The first is when you try to predict a continuous value. This is considered a regression problem. This means that your target factor contains continuous qualities like home prices. The second is when you're trying to predict a categorical value. This is considered a classification problem.

→ Features Matrix and Target Vector

The first thing you have to understand is what Scikit-Learn expects for Features Matrices and target vectors. In scikit-learn, a features matrix is a two-dimensional grid of data where rows represent samples and columns represent features. A target vector is usually one dimensional and in the case of supervised learning, what you want to predict from the data.



Let's see an example of this. the image is a pandas dataframe of the first 5 rows of the iris dataset. A single flower represent one row of the dataset and the flower measurements are the columns. In this dataset, the species column is what you are trying to predict.

	sepal_length	sepal_width	petal_length	petal_width	species
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

Let's now go over how to make sure your data is in an acceptable format

→ Import Libraries

```
# All the imports
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
from sklearn.datasets import load_iris, load_boston
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
import numpy as np
import seaborn as sns
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn import metrics
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree
from sklearn.ensemble import BaggingRegressor
from sklearn.ensemble import BaggingRegressor
```

→ Load the Dataset:

The Iris dataset is one of datasets scikit-learn comes with that do not require the downloading of any file from some external website. The code below loads the iris dataset.

```
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['species'] = data.target
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	species
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

→ Arrange Data into Features Matrix and Target Vector

```
feature_names = ['sepal length (cm)',
                 'sepal width (cm)',
                 'petal length (cm)',
                 'petal width (cm)']
```

```
# Multiple column features matrix to convert to NumPy Array
df.loc[:, feature_names]
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2
...	...	...	...	...
145	6.7	3.0	5.2	2.3
146	6.3	2.5	5.0	1.9
147	6.5	3.0	5.2	2.0
148	6.2	3.4	5.4	2.3
149	5.9	3.0	5.1	1.8

150 rows × 4 columns

```
# Convert to numpy array
x = df.loc[:, feature_names].values
```

```
# Make sure NumPy array is two dimensional
x.shape
```

```
(150, 4)
```

```
# Pandas series to convert to NumPy Array to serve as the target vector
df.loc[:, 'species']
```

```
0      0
1      0
2      0
3      0
4      0
      ..
145    2
146    2
147    2
148    2
149    2
Name: species, Length: 150, dtype: int64
```

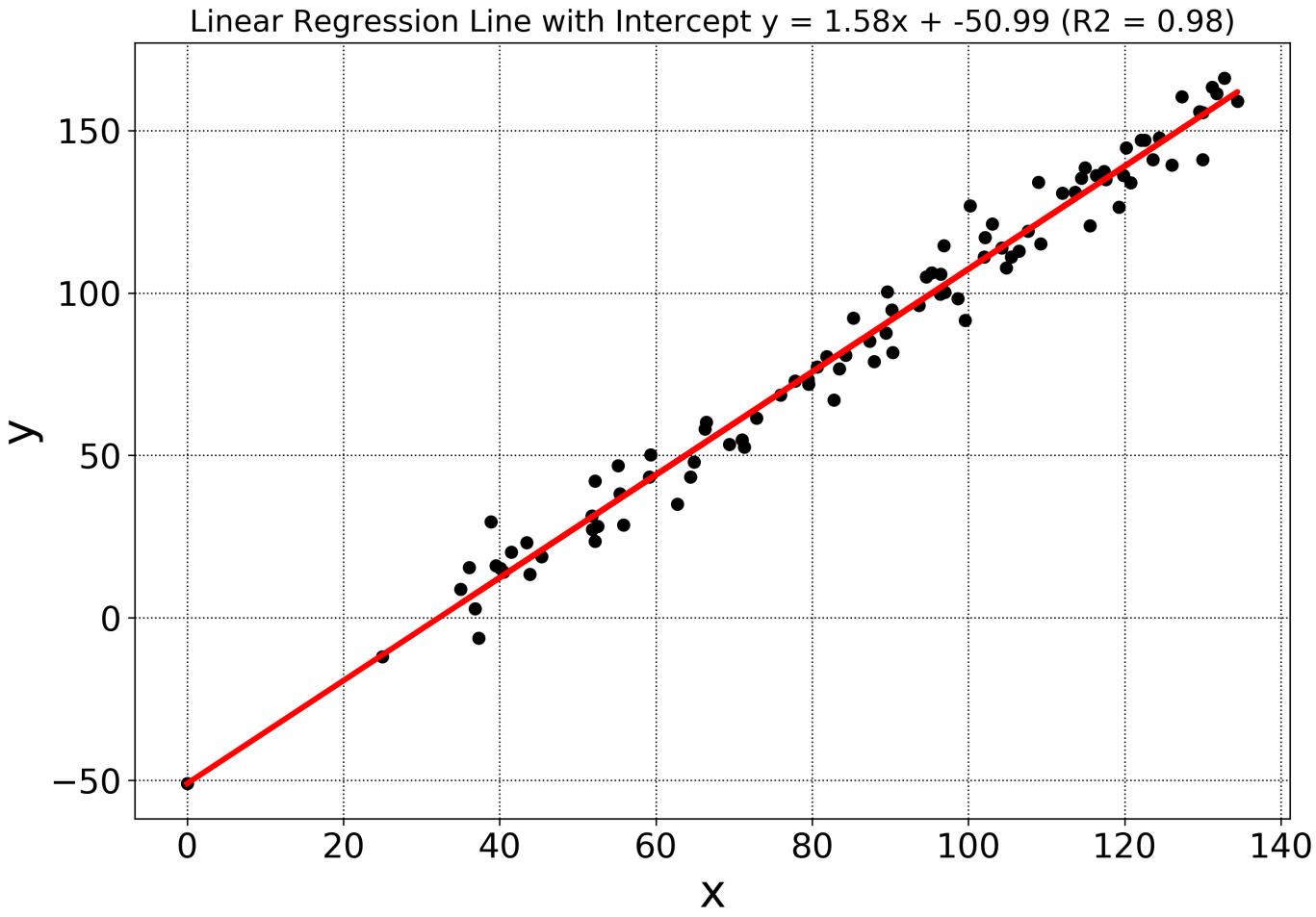
```
# using .values to convert all (:) of the column with the column name 'species'
# to a Numpy array

y = df.loc[:, 'species'].values
```

```
y.shape
```

```
(150,)
```

In this image, we see a best fit line for a bunch of points.

How do you create a complex model using scikit-learn? An easy solution is to start with a simple model like linear regression and go from there.



In this video, I'll show you can create a linear regression model using Scikit-Learn so that more complex models will be easier to create

Load the Dataset: The dataset that is loaded below is a dataset which is designed to show that Scikit-Learn requires data to be free of missing values. If you don't remove or impute your missing values, you will get an error. The goal of this dataset is to use the feature column x to predict the target column y.

```
df = pd.read_csv("https://www.evanmarie.com/content/files/class_files/scikit-learn-lile
df.head()
```

	x	y
0	0.000000	-51.000000
1	25.000000	-12.000000
2	117.583220	134.907414
3	108.922466	134.085180
4	69.887445	NaN

⇨ Remove Missing or Impute Values

If you want to build models with your data, null values are (almost) never allowed. It is important to always see how many samples have missing values and for which columns.

```
# Look at the shape of the dataframe
df.shape
```

```
(102, 2)
```

```
# There are missing values in the y column which is what we will predict
df.isnull().sum()
```

```
x    0
y    8
dtype: int64
```

You can either remove rows where there is a missing value or you can fill in missing values. The option used in this notebook is to remove rows with missing values.

```
# Remove entire rows from dataframe if they contain any nans in them or 'all'
# this may not be the best strategy for our dataset
df = df.dropna(how = 'any')
```

```
# There are no more missing values
df.isnull().sum()
```

```
x    0
y    0
dtype: int64
```

```
df.shape
```

```
(94, 2)
```

You could have filled in missing values using the `fillna` method on a pandas series if you want

Arrange Data into Features Matrix and Target Vector

```
# Convert x column to numpy array
X = df.loc[:, ['x']].values
```

```
# Features Matrix needs to be at 2 dimensional
X.shape
```

```
(94, 1)
```

```
y = df.loc[:, 'y'].values
```

```
y.shape
```

```
(94,)
```

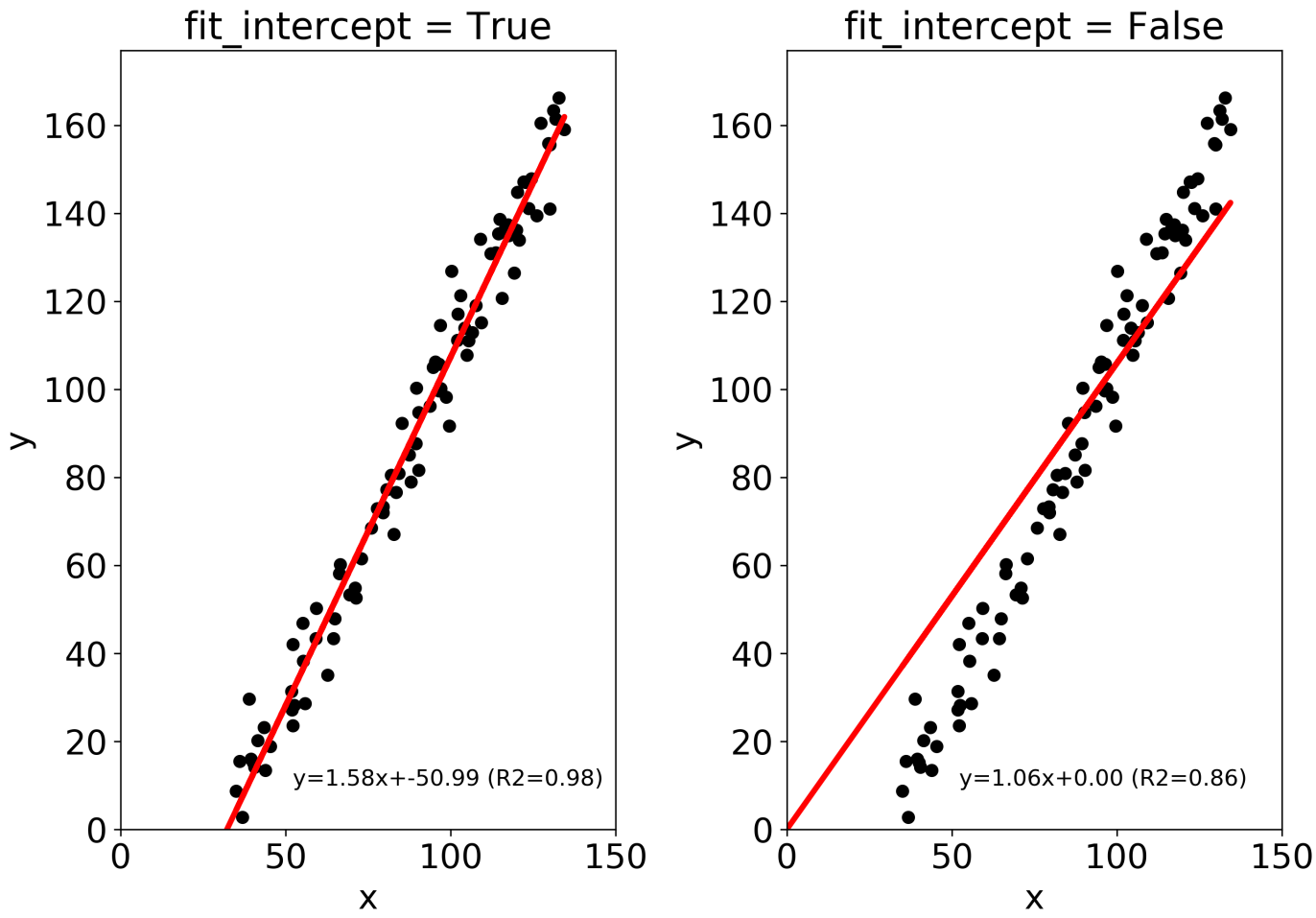
⇨ Linear Regression

Step 1: Import the model you want to use

In sklearn, all machine learning models are implemented as Python classes

Step 2: Make an instance of the Model

This is a place where you can tune the hyperparameters of a model. In the case of linear regression, you can set `fit_intercept` to True or False depending on your needs. This is an important concept as more complex models have a lot more you can tune.




```
# Make a linear regression instance
reg = LinearRegression(fit_intercept=True)
```

```
# If you want to see what you can tune for a model, you can use the help function
#help(LinearRegression)
```

Step 3: Training the model on the data, storing the information learned from the data

Model is learning the relationship between x and y

```
reg.fit(X,y)
```

```
LinearRegression()
```

Step 4: Predict the values of new data. Uses the information the model learned during the model training process

Predict for One Observation

```
# Input needs to be two dimensional (reshape makes input two dimensional )
reg.predict(X[0].reshape(-1,1))
```

```
array([-50.99119328])
```

Predict for Multiple Observations at Once

```
reg.predict(X[0:10])
```

```
array([-50.99119328, -11.39905237, 135.223663  , 121.50775193,
       102.37289634,  31.0056196 ,   4.46431068,  74.84474012,
        20.82088826,  72.16749711])
```

⇨ Measuring Model Performance

Unlike classification models where a common metric is accuracy, regression models use other metrics like R^2, the coefficient of determination to quantify your model's performance. The best possible score is 1.0. A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

```
score = reg.score(X, y)
print(score)
```

```
0.979881836115762
```

→ What is the equation of the line for the regression?

After you fit an instance of a model in scikit-learn, you can use additional attributes.

```
reg.coef_
```

```
array([1.58368564])
```

```
reg.intercept_
```

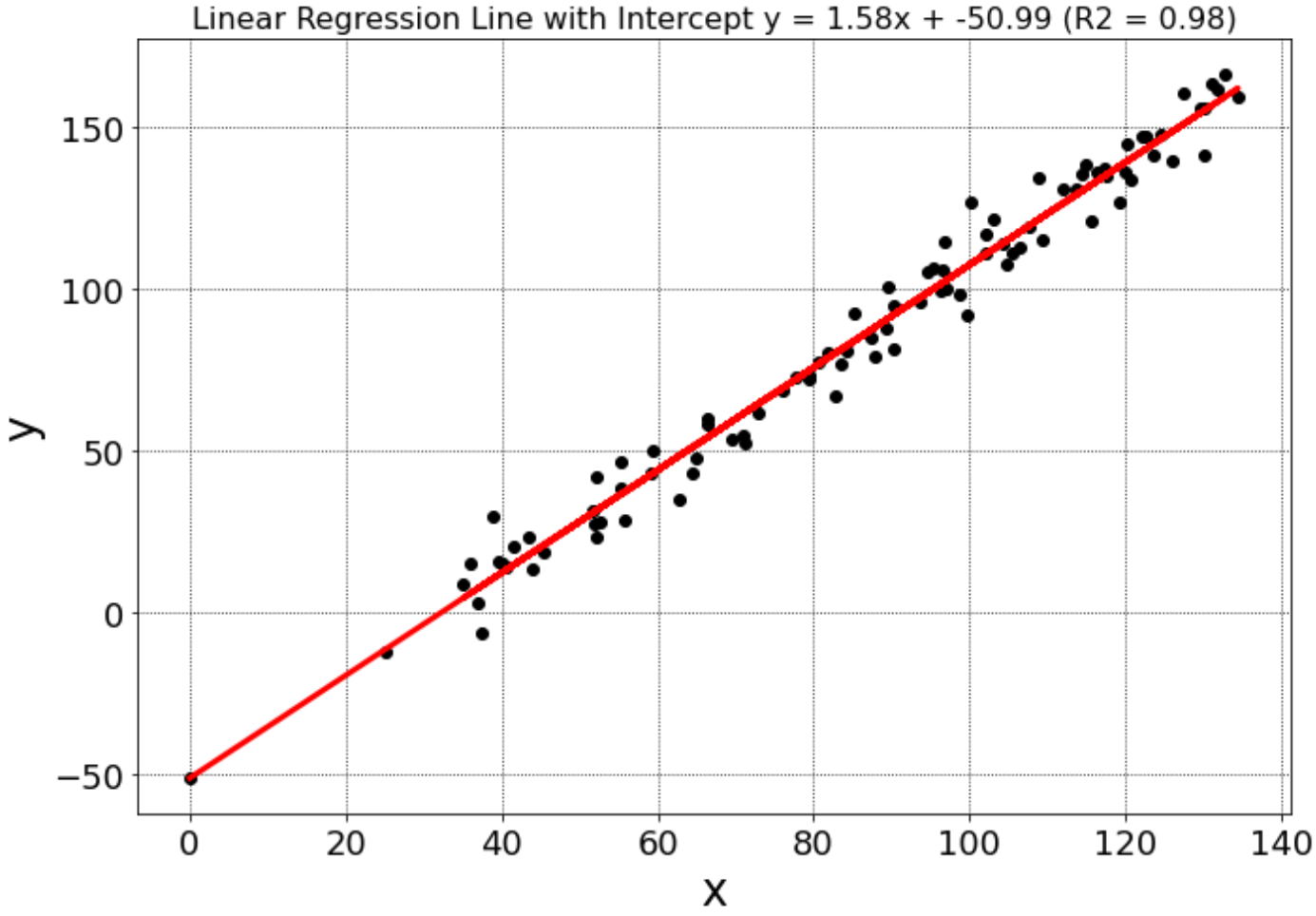
```
-50.99119328333394
```

```
m = reg.coef_[0]

b = reg.intercept_

# following slope intercept form
print("formula: y = {:.2f}x + {:.2f}".format(m, b) )
```

```
formula: y = 1.58x + -50.99
```

→ Plotting the Best Fit Linear Regression Line in Red

```
fig, ax = plt.subplots(nrows = 1, ncols = 1, figsize = (10,7));

ax.scatter(X, y, color='black');
ax.plot(X, reg.predict(X), color='red',linewidth=3);
ax.grid(True,
        axis = 'both',
        zorder = 0,
        linestyle = ':',
        color = 'k')
ax.tick_params(labelsize = 18)
ax.set_xlabel('x', fontsize = 24)
ax.set_ylabel('y', fontsize = 24)
ax.set_title("Linear Regression Line with Intercept y = {:.2f}x + {:.2f} (R2 = {:.2f})"
fig.tight_layout()
#fig.savefig('images/linearregression', dpi = 300)
```



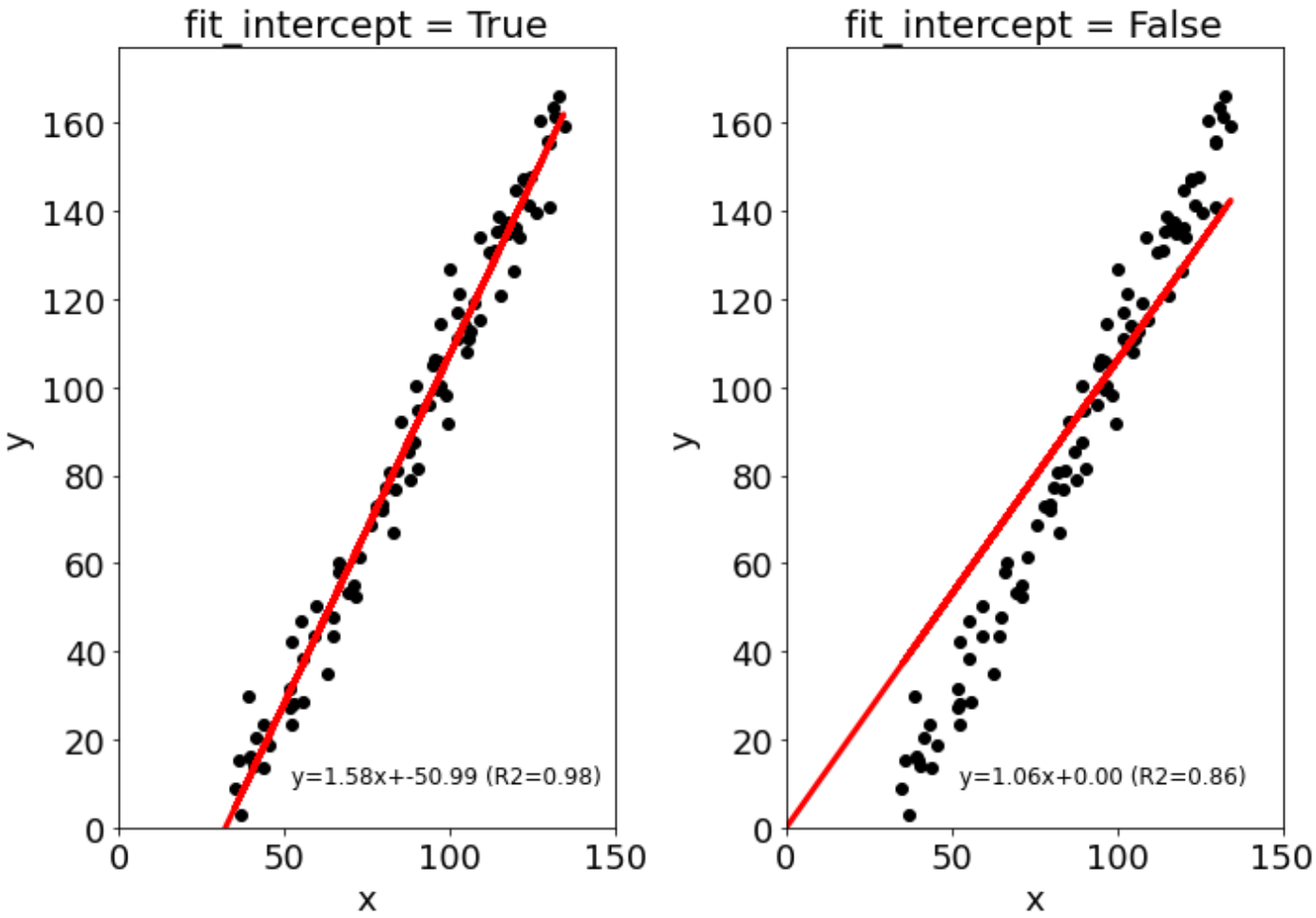
→ Plotting Models With or Without **Intercept**

In this section, you will see how changing a hyperparameter value can have a drastic impact on the R2

```
# Model with Intercept (like earlier in notebook)
reg_inter = LinearRegression(fit_intercept=True)
reg_inter.fit(X,y)
```

```
predictions_inter = reg_inter.predict(X)
score_inter = reg_inter.score(X, y)
```

```
fig, ax = plt.subplots(nrows = 1, ncols = 2, figsize = (10,7));

for index, model in enumerate([LinearRegression(fit_intercept=True), LinearRegression(f
    model.fit(X,y)
    predictions = model.predict(X)
    score = model.score(X, y)
    m = model.coef_[0]
    b = model.intercept_

    ax[index].scatter(X, y, color='black');
    ax[index].plot(X, model.predict(X), color='red',linewidth=3);

    ax[index].tick_params(labelsize = 18)
    ax[index].set_xlabel('x', fontsize = 18)
    ax[index].set_ylabel('y', fontsize = 18)
    ax[index].set_xlim(left = 0, right = 150)
    ax[index].set_ylim(bottom = 0)

    ax[index].text(50, 10, " y={:.2f}x+{:.2f} (R2={:.2f})".format(m, b, score), fontsiz

ax[0].set_title('fit_intercept = True', fontsize = 20)
ax[1].set_title('fit_intercept = False',  fontsize = 20)
fig.tight_layout()
```



A goal of supervised learning is to **build a model that performs well on new data**. If you have new data, you could see how your model performs on it. The problem is that you may not have new data, but you can simulate this experience with a train test split. In this video, I'll show you how train test split works in Scikit-Learn.

⇨ What is `train_test_split`?

1. Split the dataset into two pieces: a **training set** and a **testing set**. Typically, about 75% of the data goes to your training set and 25% goes to your test set.
2. Train the model on the **training set**.
3. Test the model on the **testing set** and evaluate the performance

Load the Dataset: The boston house-price dataset is one of datasets scikit-learn comes with that do not require the downloading of any file from some external website. The code below loads the boston dataset.

```
data = load_boston()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
df.head()
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/utils/deprecation.py:87: FutureWarning:
Function load_boston is deprecated; `load_boston` is deprecated in 1.0 and will be
removed in 1.2.

    The Boston housing prices dataset has an ethical problem. You can refer to
```

```
  the documentation of this function for further details.

  The scikit-learn maintainers therefore strongly discourage the use of this
  dataset unless the purpose of the code is to study and educate about
  ethical issues in data science and machine learning.

  In this special case, you can fetch the dataset from the original
  source::

      import pandas as pd
      import numpy as np


      data_url = "http://lib.stat.cmu.edu/datasets/boston"
      raw_df = pd.read_csv(data_url, sep="\s+", skiprows=22, header=None)
      data = np.hstack([raw_df.values[::2, :], raw_df.values[1::2, :2]])
      target = raw_df.values[1::2, 2]

  Alternative datasets include the California housing dataset (i.e.
  :func:`~sklearn.datasets.fetch_california_housing`) and the Ames housing
  dataset. You can load the datasets as follows::

      from sklearn.datasets import fetch_california_housing
      housing = fetch_california_housing()

  for the California housing dataset and::

      from sklearn.datasets import fetch_openml
      housing = fetch_openml(name="house_prices", as_frame=True)

  for the Ames housing dataset.

warnings.warn(msg, category=FutureWarning)
```

	CRIM	ZN	INDUS	CHAS	NOX	RM	AGE	DIS	RAD	TAX	PTRATIO	B	LSTAT	target
0	0.00632	18.0	2.31	0.0	0.538	6.575	65.2	4.0900	1.0	296.0	15.3	396.90	4.98	24.0
1	0.02731	0.0	7.07	0.0	0.469	6.421	78.9	4.9671	2.0	242.0	17.8	396.90	9.14	21.6
2	0.02729	0.0	7.07	0.0	0.469	7.185	61.1	4.9671	2.0	242.0	17.8	392.83	4.03	34.7
3	0.03237	0.0	2.18	0.0	0.458	6.998	45.8	6.0622	3.0	222.0	18.7	394.63	2.94	33.4
4	0.06905	0.0	2.18	0.0	0.458	7.147	54.2	6.0622	3.0	222.0	18.7	396.90	5.33	36.2

```
X = df.loc[:, ['RM', 'LSTAT', 'PTRATIO']].values
```

```
y = df.loc[:, 'target'].values
```

→ Train Test Split

RM	LSTAT	PTRATIO	target
6.575000	4.980000	15.300000	24.000000
6.421000	9.140000	17.800000	29.000000
7.185000	4.030000	17.800000	22.900000
6.998000	2.940000	18.700000	10.200000
7.147000	5.330000	18.700000	14.800000
6.430000	5.210000	18.700000	32.400000
6.012000	12.430000	15.200000	43.100000
6.172000	19.150000	15.200000	33.400000
5.631000	29.930000	15.200000	10.400000
6.004000	17.100000	15.200000	17.800000

X_train

X_test

y_train

y_test

The colors in the image indicate which variable (X_train, X_test, y_train, y_test) the data from the dataframe df went to for a particular train test split (not necessarily the exact split of the code below).

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=3)
```

→ Linear Regression Model

```
# Make a linear regression instance
reg = LinearRegression(fit_intercept=True)

# Train the model on the training set.
reg.fit(X_train, y_train)
```

```
LinearRegression()
```

→ Measuring Model Performance

By measuring model performance on the test set, you can estimate how well your model is likely to perform on new data (out-of-sample data)

```
# Test the model on the testing set and evaluate the performance
score = reg.score(X_test, y_test)
print(score)
```

```
0.7155620757319656
```

⇨ LOGISTIC REGRESSION: for binary classification

How do you create a logistic regression model using Scikit-Learn? The first thing you need to know is that despite the name logistic regression containing the word regression, logistic regression is a model used for classification. **Classification models** can be used for tasks like classifying flower species or image recognition. All of this of course depends on the availability and quality of your data. Logistic Regression has some advantages including

- Model training and predictions are relatively fast
- No tuning is usually needed for logistic regression unless you want to regularize your model.
- Finally, it can perform well with a small number of observations.

In this video, I'll share with you **how you can create a logistic regession model for binary classification**.

Load the Dataset: The code below loads a modified version of the iris dataset which has two classes. A 1 is a virginica flower and a 0 is versicolor flower.

```
df = pd.read_csv('https://www.evanmarie.com/content/files/class_files/scikit-learn-lile
```

```
df.shape
```

```
(100, 5)
```

→ Splitting Data into Training and Test Sets

```
X_train, X_test, y_train, y_test = train_test_split(df[['petal length (cm)']], df['targ
```

→ Standardize the Data

Logistic Regression is effected by scale so you need to scale the features in the data before using Logistic Regresison. You can transform the data onto unit scale (mean = 0 and variance = 1) for better performance. Scikit-Learn's `StandardScaler` helps standardize the dataset's features. **Note you fit on the training set and transform on the training and test set.**

```
scaler = StandardScaler()

# Fit on training set only.
scaler.fit(X_train)

# Apply transform to both the training set and the test set.
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

→ Logistic Regression

Step 1: Import the model you want to use

In sklearn, all machine learning models are implemented as Python classes

Step 2: Make an instance of the Model

This is a place where we can tune the hyperparameters of a model. Typically this is where you tune C is related to regularization

```
clf = LogisticRegression()
```

Step 3: Training the model on the data, storing the information learned from the data

Model is learning the relationship between x (features sepal width, sepal height etc) and y (labels-which species of iris)

```
clf.fit(X_train, y_train)
```

```
LogisticRegression()
```

Step 4: Predict the labels of new data (new flowers)

Logistic regression also allows you to see prediction probabilities as well as a prediction. This is not like other algorithms like decision trees for classification which only give you a prediction not a probability.

```
# One observation's petal length after standardization
X_test[0].reshape(1,-1)
```

```
array([[-0.12093628]])
```

```
print('prediction', clf.predict(X_test[0].reshape(1,-1))[0])
print('probability', clf.predict_proba(X_test[0].reshape(1,-1)))
```

```
prediction 0
probability [[0.52720087 0.47279913]]
```

If this is unclear, let's visualize how logistic regression makes predictions by looking at our test data!

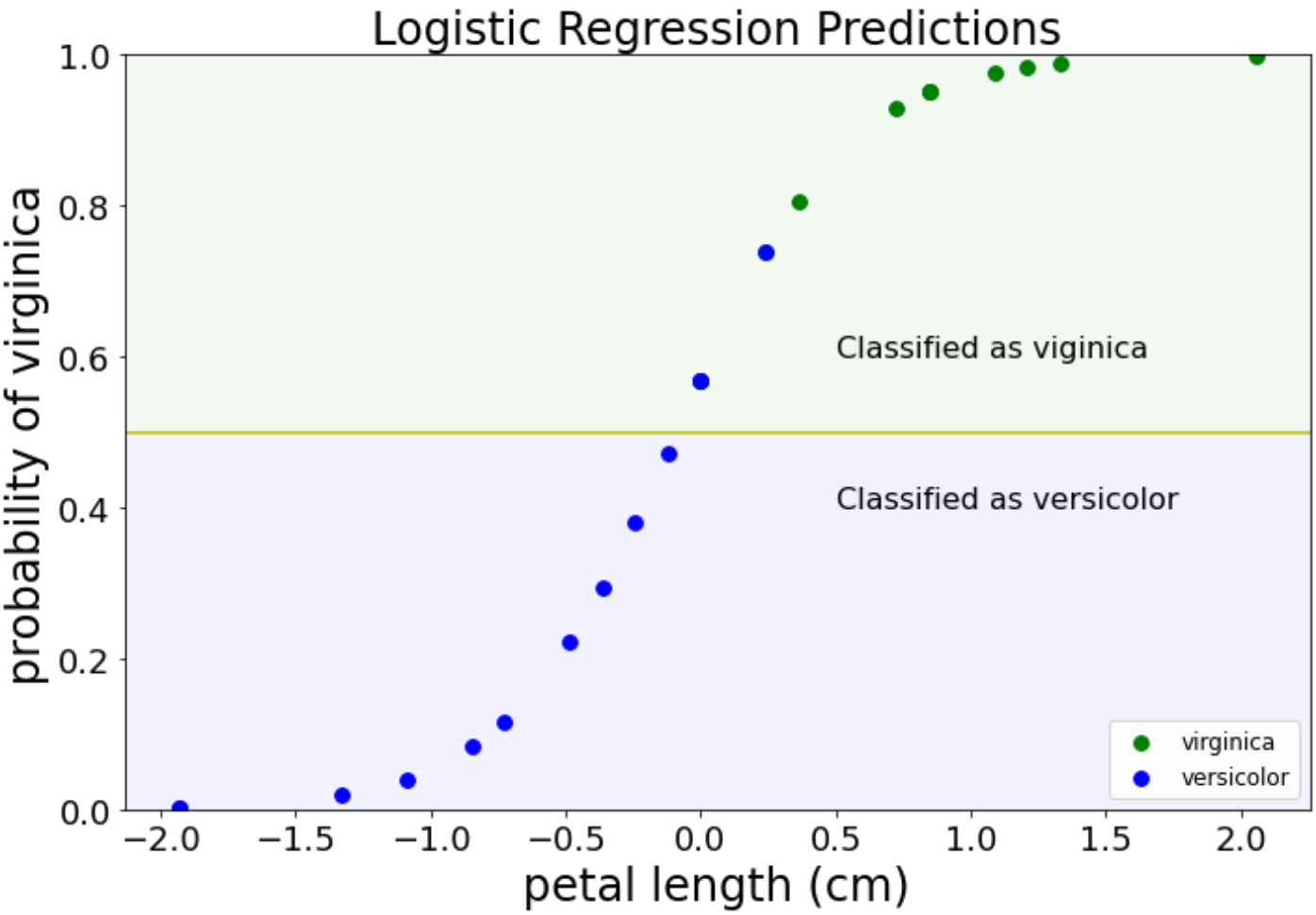
```
example_df = pd.DataFrame()
example_df.loc[:, 'petal length (cm)'] = X_test.reshape(-1)
example_df.loc[:, 'target'] = y_test.values
example_df['logistic_preds'] = pd.DataFrame(clf.predict_proba(X_test))[1]
```

```
example_df.head()
```

	petal length (cm)	target	logistic_preds
0	-0.120936	0	0.472799

	petal length (cm)	target	logistic_preds
1	0.846554	1	0.950658
2	0.000000	0	0.568197
3	2.055917	1	0.998879
4	1.330299	1	0.988926

```
fig, ax = plt.subplots(nrows = 1, ncols = 1, figsize = (10,7));


virginicaFilter = example_df['target'] == 1
versicolorFilter = example_df['target'] == 0

ax.scatter(example_df.loc[virginicaFilter, 'petal length (cm)'].values,
            example_df.loc[virginicaFilter, 'logistic_preds'].values,
           color = 'g',
           s = 60,
           label = 'virginica')


ax.scatter(example_df.loc[versicolorFilter, 'petal length (cm)'].values,
            example_df.loc[versicolorFilter, 'logistic_preds'].values,
           color = 'b',
           s = 60,
           label = 'versicolor')

ax.axhline(y = .5, c = 'y')

ax.axhspan(.5, 1, alpha=0.05, color='green')
ax.axhspan(0, .4999, alpha=0.05, color='blue')
ax.text(0.5, .6, 'Classified as viginica', fontsize = 16)
ax.text(0.5, .4, 'Classified as versicolor', fontsize = 16)

ax.set_ylim(0,1)
ax.legend(loc = 'lower right', markerscale = 1.0, fontsize = 12)
ax.tick_params(labelsize = 18)
ax.set_xlabel('petal length (cm)', fontsize = 24)
ax.set_ylabel('probability of virginica', fontsize = 24)
ax.set_title('Logistic Regression Predictions', fontsize = 24)
fig.tight_layout()
```



→ Measuring Model Performance

While there are other ways of measuring model performance (precision, recall, F1 Score, ROC Curve, etc), let's keep this simple and use accuracy as our metric. To do this are going to see how the model performs on new data (test set)

Accuracy is defined as: (fraction of correct predictions): correct predictions / total number of data points

```
score = clf.score(X_test, y_test)
print(score)
```

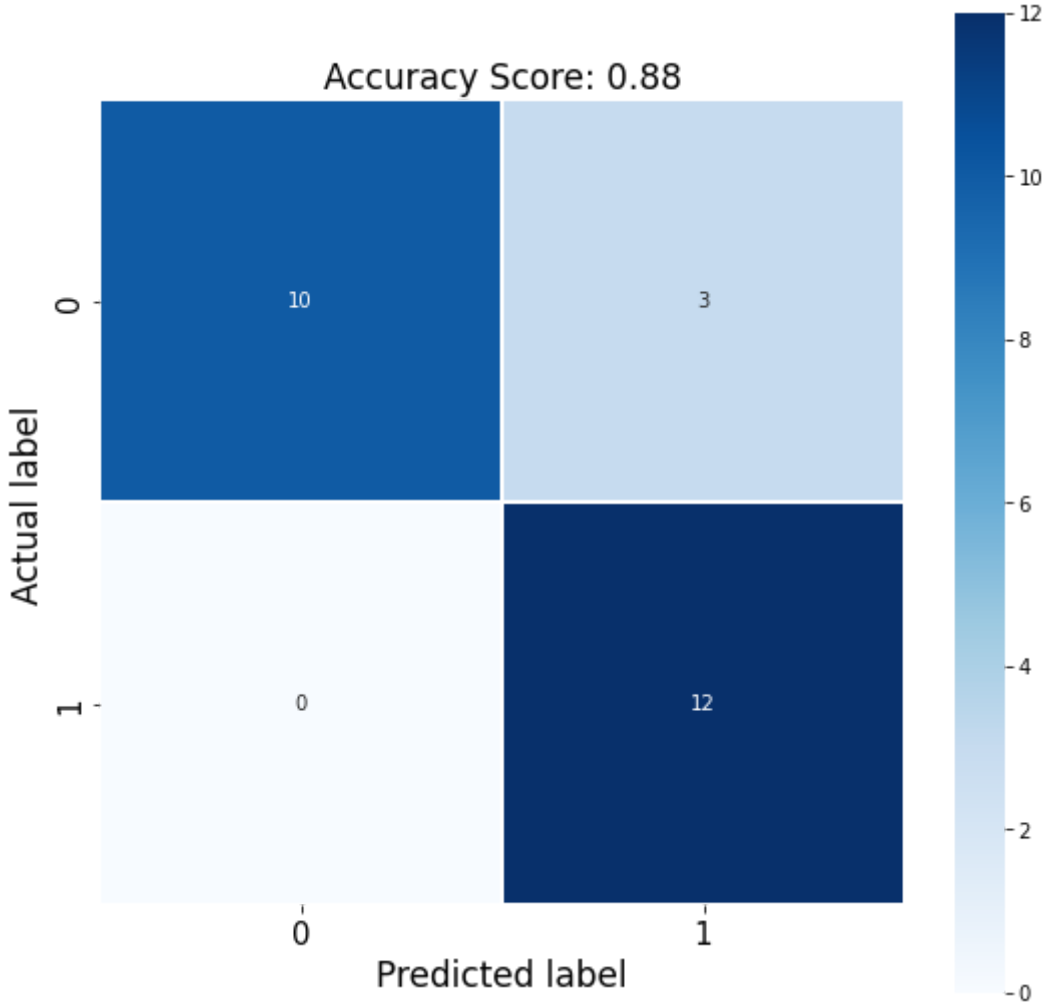
```
0.88
```

⇨ Confusion Matrix:

Accuracy is one metric, but it doesn't say give much insight into what was wrong. Let's look at a confusion matrix

In the field of machine learning and specifically the problem of statistical classification, a confusion matrix, also known as an error matrix,[10] is a specific table layout that allows visualization of the performance of an algorithm, typically a supervised learning one (in unsupervised learning it is usually called a matching matrix). Each row of the matrix represents the instances in an actual class while each column represents the instances in a predicted class, or vice versa – both variants are found in the literature.[11] The name stems from the fact that it makes it easy to see whether the system is confusing two classes (i.e. commonly mislabeling one as another). (Wikipedia)

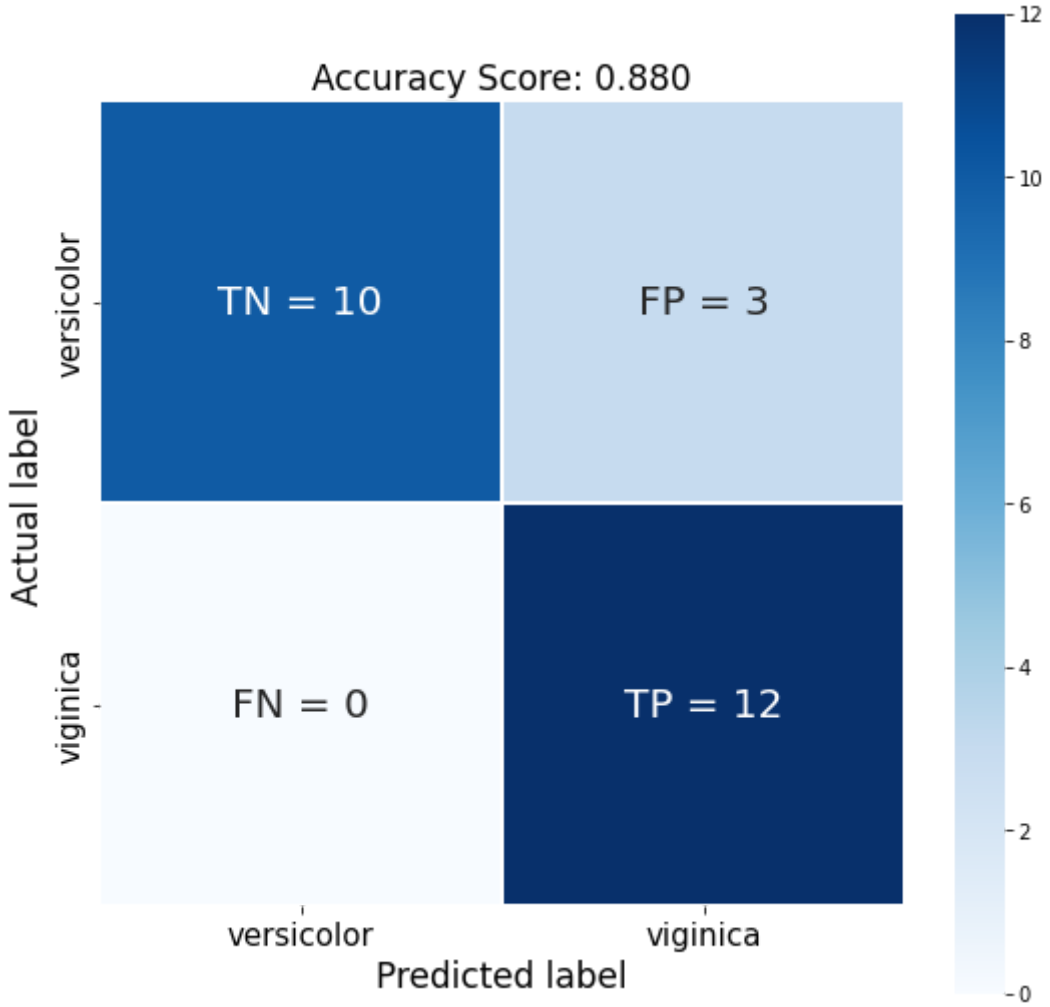
```
cm = metrics.confusion_matrix(y_test, clf.predict(X_test))

plt.figure(figsize=(9,9))
sns.heatmap(cm, annot=True,
            fmt=".0f",
            linewidths=.5,
            square = True,
            cmap = 'Blues');
plt.ylabel('Actual label', fontsize = 17);
plt.xlabel('Predicted label', fontsize = 17);
plt.title('Accuracy Score: {}'.format(score), size = 17);
plt.tick_params(labelsize= 15)
```



Let's look at the same information in a table in a clearer way.

```
# ignore this code

modified_cm = []
for index,value in enumerate(cm):
    if index == 0:
        modified_cm.append(['TN = ' + str(value[0]), 'FP = ' + str(value[1])])
    if index == 1:
        modified_cm.append(['FN = ' + str(value[0]), 'TP = ' + str(value[1])])
```

```
plt.figure(figsize=(9,9))
sns.heatmap(cm, annot=np.array(modified_cm),
            fmt="",
            annot_kws={"size": 20},
            linewidths=.5,
            square = True,
            cmap = 'Blues',
            xticklabels = ['versicolor', 'viginica'],
            yticklabels = ['versicolor', 'viginica'],
            );

plt.ylabel('Actual label', fontsize = 17);
plt.xlabel('Predicted label', fontsize = 17);
plt.title('Accuracy Score: {:.3f}'.format(score), size = 17);
plt.tick_params(labelsize= 15)
```



A lot of classification models like logistic regression were originally designed for binary classification, that is predicting whether something is one thing or another. For datasets with more than two classes, what do you do? For multiclass classification problems, one approach is to split the task into multiple binary classification datasets and fit a binary classification model on each. In this video, we will explore the One-vs-Rest strategy and how you can apply it to logisitic regression using Scikit-Learn.

⇨ One-vs-Rest

One Versus Rest (OvR), which is also called **one versus all (OvA)** is a technique that extends binary classifiers to multi-class problems. Here is how it works:

- **You train one classifier per class, where one class is treated as the positive class and the other classes are considered negative classes.**

For example, say you have an image recogition task. Your dataset has 4 classes, the digits 0, 1, 2, and 3. Your goal is to classify them. Using the one versus rest approach, you break down the task into 4 binary classification problems.

> Binary Classification Problem 1: digit 0 vs digits 1, 2, and 3

> Binary Classification Problem 2: digit 1 vs digits 0, 2, and 3

> Binary Classification Problem 3: digit 2 vs digits 0, 1, and 3

> Binary Classification Problem 4: digit 3 vs digits 0, 1, and 2

From there, if you want to classify a new sample, you would use each of the classifiers. The model that predicts the highest class probability is the predicted class.

Load the Dataset: The code below loads a modified version of the digits dataset which is arranged in a csv file for convenience. **The data consists of pixel intensity values for 720 images that are 8 by 8 pixels**. Each image is labeled with a number from 0-4.

```
df = pd.read_csv('https://www.evanmarie.com/content/files/class_files/scikit-learn-lile
```

```
df.head()
```

	0	1	2	3	4	5	6	7	8	9	...	55	56	57	58	59	60	61	62	63	label
0	0	0	5	13	9	1	0	0	0	0	...	0	0	0	6	13	10	0	0	0	0
1	0	0	0	12	13	5	0	0	0	0	...	0	0	0	0	11	16	10	0	0	1
2	0	0	0	4	15	12	0	0	0	0	...	0	0	0	0	3	11	16	9	0	2
3	0	0	7	15	13	1	0	0	0	8	...	0	0	0	7	13	13	9	0	0	3
4	0	0	1	9	15	11	0	0	0	0	...	0	0	0	1	10	13	3	0	0	0

5 rows × 65 columns

```
df.shape
```

```
(720, 65)
```

→ Visualize Each Digit

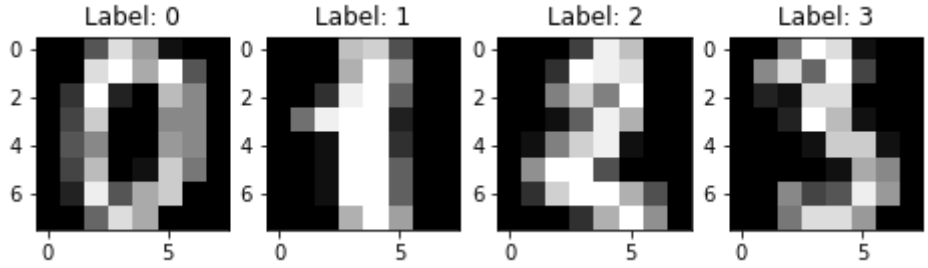
```
pixel_colnames = df.columns[:-1]
```

```
pixel_colnames
```

```
Index(['0', '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12',
       '13', '14', '15', '16', '17', '18', '19', '20', '21', '22', '23', '24',
       '25', '26', '27', '28', '29', '30', '31', '32', '33', '34', '35', '36',
       '37', '38', '39', '40', '41', '42', '43', '44', '45', '46', '47', '48',
       '49', '50', '51', '52', '53', '54', '55', '56', '57', '58', '59', '60',
       '61', '62', '63'],
      dtype='object')
```

```
# Get all columns except the label column for the first image
image_values = df.loc[0, pixel_colnames].values
```

```
plt.figure(figsize=(10,2))
for index in range(0, 4):

    plt.subplot(1, 5, 1 + index )
    image_values = df.loc[index, pixel_colnames].values
    image_label = df.loc[index, 'label']
    plt.imshow(image_values.reshape(8,8), cmap ='gray')
    plt.title('Label: ' + str(image_label))
```



→ Splitting Data into Training and Test Sets

```
X_train, X_test, y_train, y_test = train_test_split(df[pixel_colnames], df['label'], ra
```

→ Standardize the Data

Logistic Regression is effected by scale so you need to scale the features in the data before using Logistic Regresison. You can transform the data onto unit scale (mean = 0 and variance = 1) for better performance. Scikit-Learn's `StandardScaler` helps standardize the dataset's features. Note you fit on the training set and transform on the training and test set.

```
scaler = StandardScaler()

# Fit on training set only.
scaler.fit(X_train)
```

```
# Apply transform to both the training set and the test set.
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

⇨ Logistic Regression

```
# multi_class is specifying one versus rest
clf = LogisticRegression(solver='liblinear',
                         multi_class='ovr',
                         random_state = 0)

clf.fit(X_train, y_train)
print('Training accuracy:', clf.score(X_train, y_train))
print('Test accuracy:', clf.score(X_test, y_test))
```

```
Training accuracy: 1.0
Test accuracy: 1.0
```

Both the training and test accuracies are very high. If you access the intercept terms by using the `intercept_` attribute, you can see that the array has four values. Since the Logistic Regression instance was fit on a multiclass dataset via the OvR approach, the first intercept belongs to the model that fits digit 0 versus digits 1,2, and 3. The second value is the intercept of the model that fits digit 1 versus digits 0,2, and 3. Etc.

```
clf.intercept_
```

```
array([-2.712674  , -3.54379096, -3.18367757, -2.623974  ])
```

Similarly, you can get 4 different coefficient matrices.

```
clf.coef_.shape
```

```
(4, 64)
```

→ Predictions

```
# The second class is the highest score so it will be the prediction for this data
clf.predict_proba(X_test[0:1])
```

```
array([[0.00183123, 0.98368966, 0.00536378, 0.00911533]])
```

```
clf.predict(X_test[0:1])
```

```
array([1])
```

Machine Learning with Scikit-learn

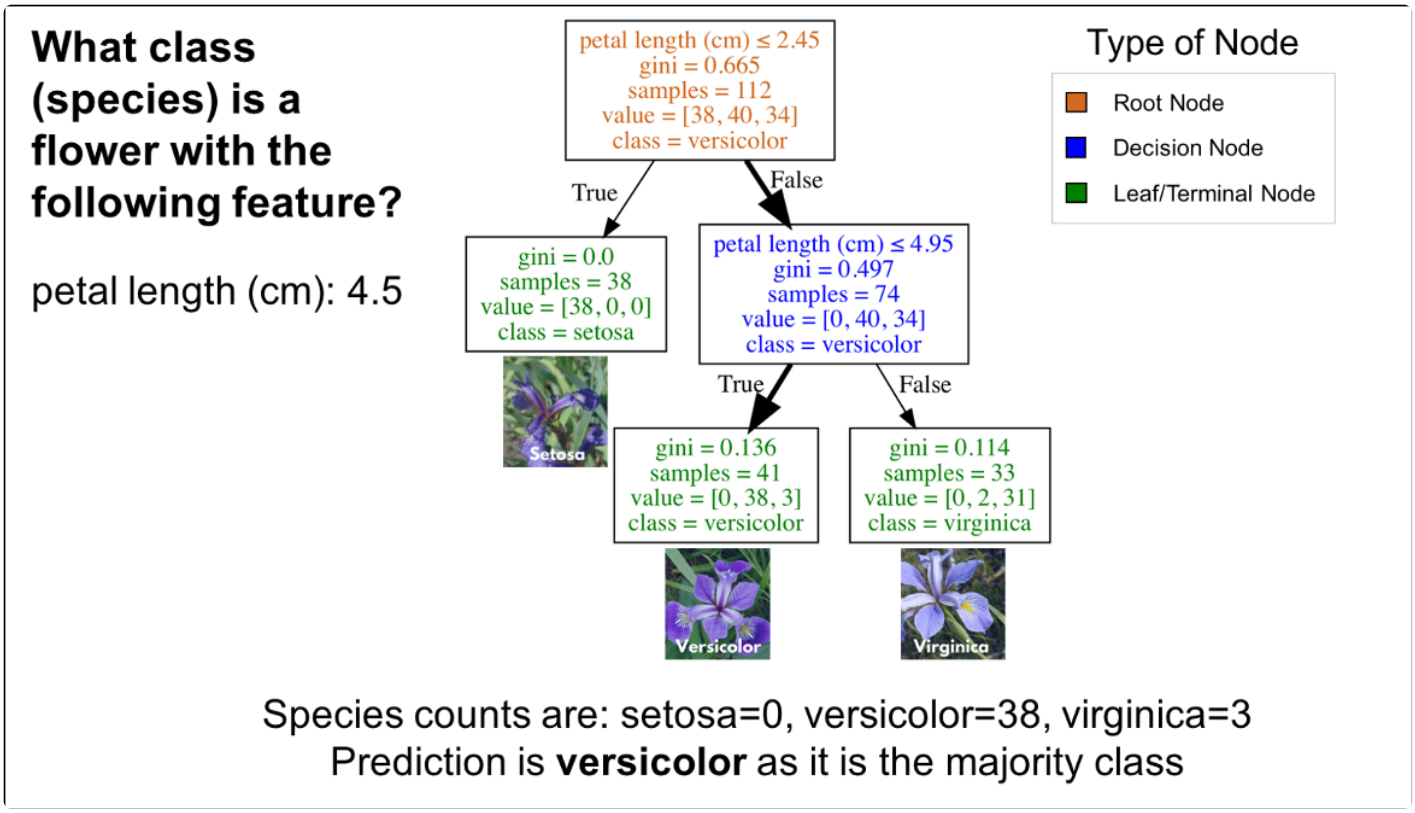
Part Two: Decision Trees and Bagging

Michael Galarnyk, LinkedIn Learning

```
# All the imports
%matplotlib inline
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import seaborn as sns


from sklearn.datasets import load_iris, load_boston
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn import metrics

from sklearn.tree import DecisionTreeClassifier

# Tree Model Visualization
from sklearn import tree
from sklearn.tree import export_text

# Random Forest Regressor and Classifier Models
from sklearn.ensemble import BaggingRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import BaggingClassifier
```

⇨ Decision Trees

One of the most important considerations when choosing a machine learning algorithm is how interpretable it is. The ability to explain how an algorithm makes predictions is useful to not only you, but also to potential stakeholders. A very interpretable machine learning algorithm is a decision tree which you can think of as a series of questions designed to assign a class or predict a continuous value depending on the task. The example image is a decision tree designed for classification.



In this video, I'll share with you how you can create and tune a decision tree using scikit-learn.

Load the Dataset: The Iris dataset is one of datasets scikit-learn comes with that do not require the downloading of any file from some external website. The code below loads the iris dataset.

```
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

→ Splitting Data into Training and Test Sets

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5	3.6	1.4	0.2	0
5	5.4	3.9	1.7	0.4	0
6	4.6	3.4	1.4	0.3	0
7	5	3.4	1.5	0.2	0
8	4.4	2.9	1.4	0.2	0
9	4.9	3.1	1.5	0.1	0

X_train
X_test
Y_train
Y_test

The colors in the image indicate which variable (X_train, X_test, Y_train, Y_test) the data from the dataframe df went to for a particular train test split (not necessarily the exact split of the code below).

```
X_train, X_test, y_train, y_test = train_test_split(df[data.feature_names], df['target'
```

Note, another benefit of Decision Trees is that you don't have to standardize your features unlike other algorithms like logistic regression and K-Nearest Neighbors.

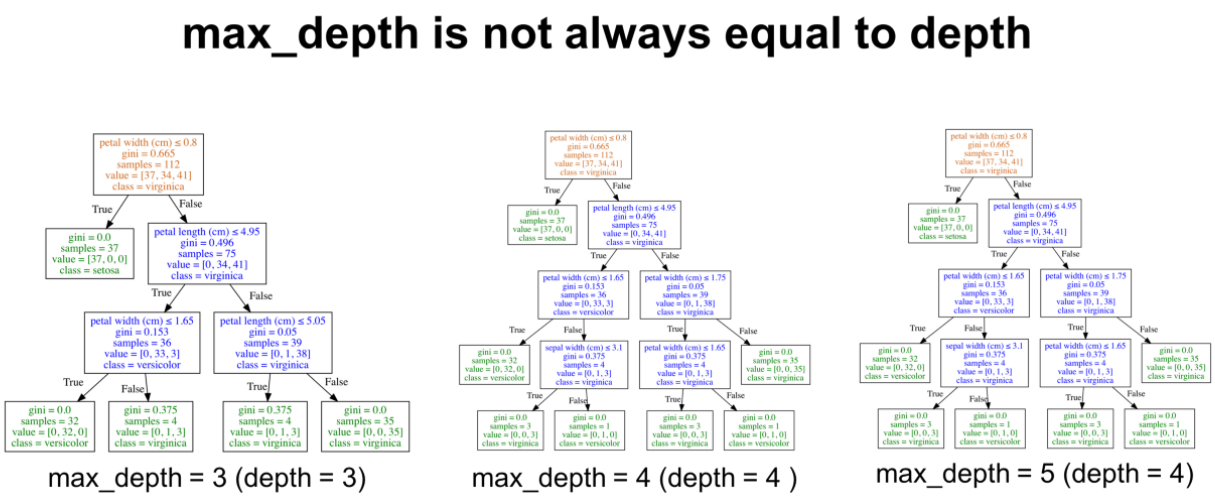
→ Decision Tree

Step 1: Import the model you want to use

In sklearn, all machine learning models are implemented as Python classes

Step 2: Make an instance of the Model

This is a place where we can tune the hyperparameters of a model. The code below constrains the model to have at most a depth of 2. Tree depth is a measure of how many splits it makes before coming to a prediction.



```
clf = DecisionTreeClassifier(max_depth = 2,
                             random_state = 0)
```

Step 3: Training the model on the data, storing the information learned from the data

Model is learning the relationship between x (features sepal width, sepal height etc) and y (labels-which species of iris)

```
clf.fit(X_train, y_train)
```

```
DecisionTreeClassifier(max_depth=2, random_state=0)
```

Step 4: Predict the labels of new data (new flowers)

Uses the information the model learned during the model training process

```
# Predict for One Observation
clf.predict(X_test.iloc[0].values.reshape(1, -1))
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/base.py:451: UserWarning: X does not have valid feature names, but DecisionTreeClassifier was fitted with feature names
  "X does not have valid feature names, but"
```

```
array([2])
```

Predict for Multiple Observations at Once

```
clf.predict(X_test[0:10])
```

```
array([2, 1, 0, 2, 0, 2, 0, 1, 1, 1])
```

→ Measuring Model Performance

While there are other ways of measuring model performance (precision, recall, F1 Score, ROC Curve, etc), we are going to keep this simple and use accuracy as our metric. To do this are going to see how the model performs on new data (test set)

Accuracy is defined as: (fraction of correct predictions): correct predictions / total number of data points

```
score = clf.score(X_test, y_test)
print(score)
```

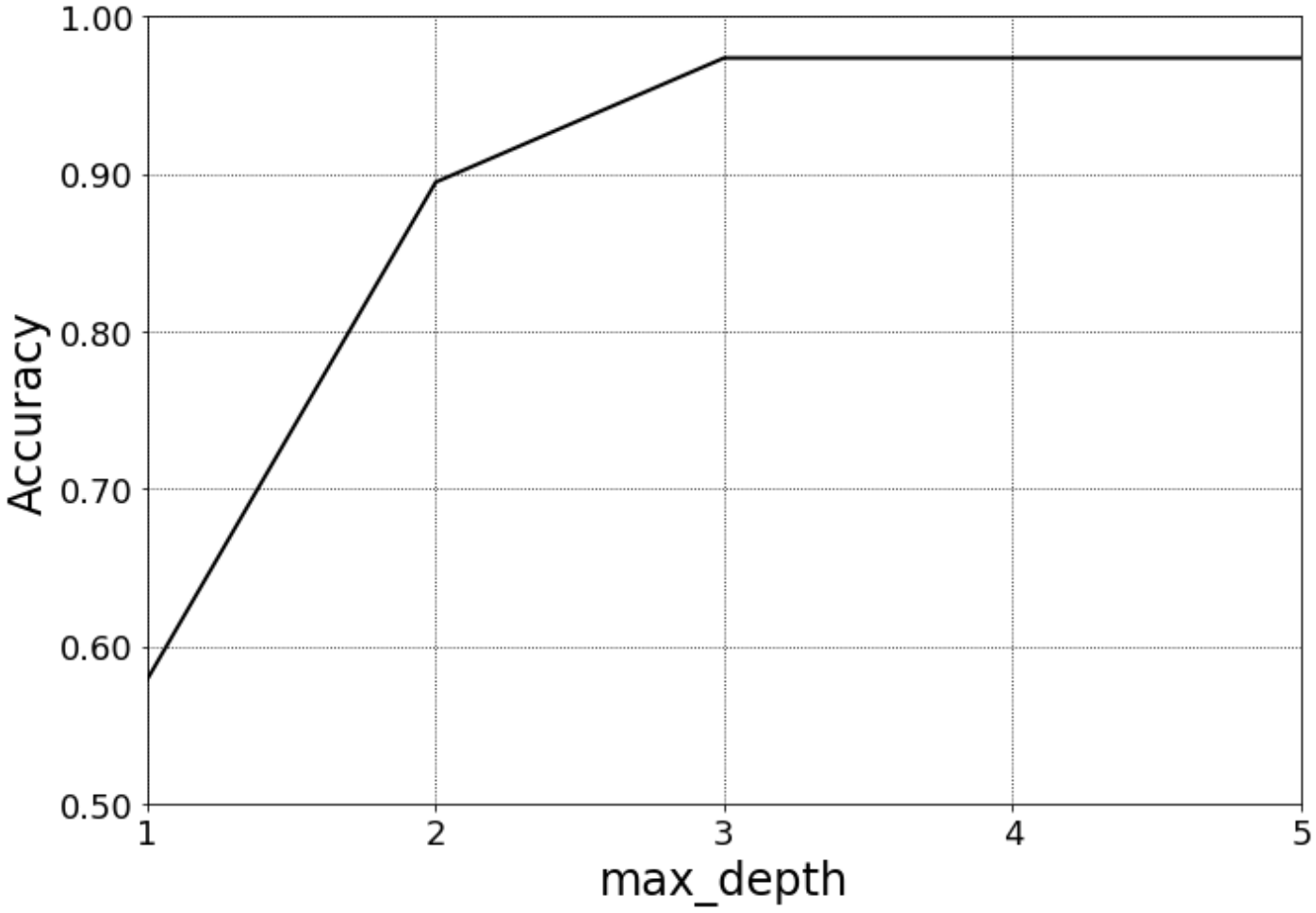
```
0.8947368421052632
```

→ Finding the Optimal max_depth

```
# List of values to try for max_depth:
max_depth_range = list(range(1, 6))

# List to store the average RMSE for each value of max_depth:
```

```
accuracy = []
for depth in max_depth_range:
    clf = DecisionTreeClassifier(max_depth = depth,
                                 random_state = 0)
    clf.fit(X_train, y_train)
    score = clf.score(X_test, y_test)
    accuracy.append(score)
```

```
fig, ax = plt.subplots(nrows = 1, ncols = 1, figsize = (10,7));

ax.plot(max_depth_range,
        accuracy,
        lw=2,
        color='k')

ax.set_xlim([1, 5])
ax.set_ylim([.50, 1.00])
ax.grid(True,
        axis = 'both',
        zorder = 0,
        linestyle = ':',
        color = 'k')

yticks = ax.get_yticks()

y_ticklist = []
for tick in yticks:
    y_ticklist.append(str(tick).ljust(4, '0')[0:4])
ax.set_yticklabels(y_ticklist)
ax.tick_params(labelsize = 18)
ax.set_xticks([1,2,3,4,5])
ax.set_xlabel('max_depth', fontsize = 24)
ax.set_ylabel('Accuracy', fontsize = 24)
fig.tight_layout()
#fig.savefig('images/max_depth_vs_accuracy.png', dpi = 300)
```



→ Visualizing Decision Trees:

How do you understand how a decision tree makes predictions? One of the strengths of decision trees are that they are relatively easy to interpret as you can make a visualization based on your model. This is not only a powerful way to understand your model, but also to communicate how your model works to stakeholders.

Load the Dataset: The Iris dataset is one of datasets scikit-learn comes with that do not require the downloading of any file from some external website. The code below loads the iris dataset.

```
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	0
1	4.9	3.0	1.4	0.2	0
2	4.7	3.2	1.3	0.2	0
3	4.6	3.1	1.5	0.2	0
4	5.0	3.6	1.4	0.2	0

→ Split Data into Training and Test Sets

```
X_train, X_test, Y_train, Y_test = train_test_split(df[data.feature_names], df['target'
```

⇨ Scikit-learn 4-Step Modeling Pattern

Step 1: Import the model you want to use

In sklearn, all machine learning models are implemented as Python classes

Step 2: Make an instance of the Model

```
clf = DecisionTreeClassifier(max_depth = 2,
                             random_state = 0)
```

Step 3: Training the model on the data, storing the information learned from the data

Model is learning the relationship between x (features: sepal width, sepal height etc) and y (labels-which species of iris)

```
clf.fit(X_train, Y_train)
```

```
DecisionTreeClassifier(max_depth=2, random_state=0)
```

Step 4: Predict the labels of new data (new flowers)

Uses the information the model learned during the model training process

```
# Predict for One Observation (image)
clf.predict(X_test.iloc[0].values.reshape(1, -1))
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/base.py:451: UserWarning: X does not
have valid feature names, but DecisionTreeClassifier was fitted with feature names
  "X does not have valid feature names, but"
```

```
array([2])
```

Predict for Multiple Observations (images) at Once

```
clf.predict(X_test[0:10])
```

```
array([2, 1, 0, 2, 0, 2, 0, 1, 1, 1])
```

→ Measuring Model Performance

Accuracy is defined as: (fraction of correct predictions): correct predictions / total number of data points

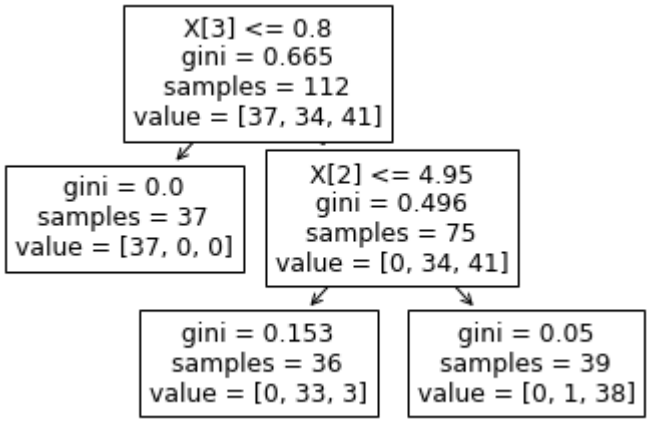
```
score = clf.score(X_test, Y_test)
print(score)
```

```
0.8947368421052632
```

→ How to Visualize Decision Trees using Matplotlib

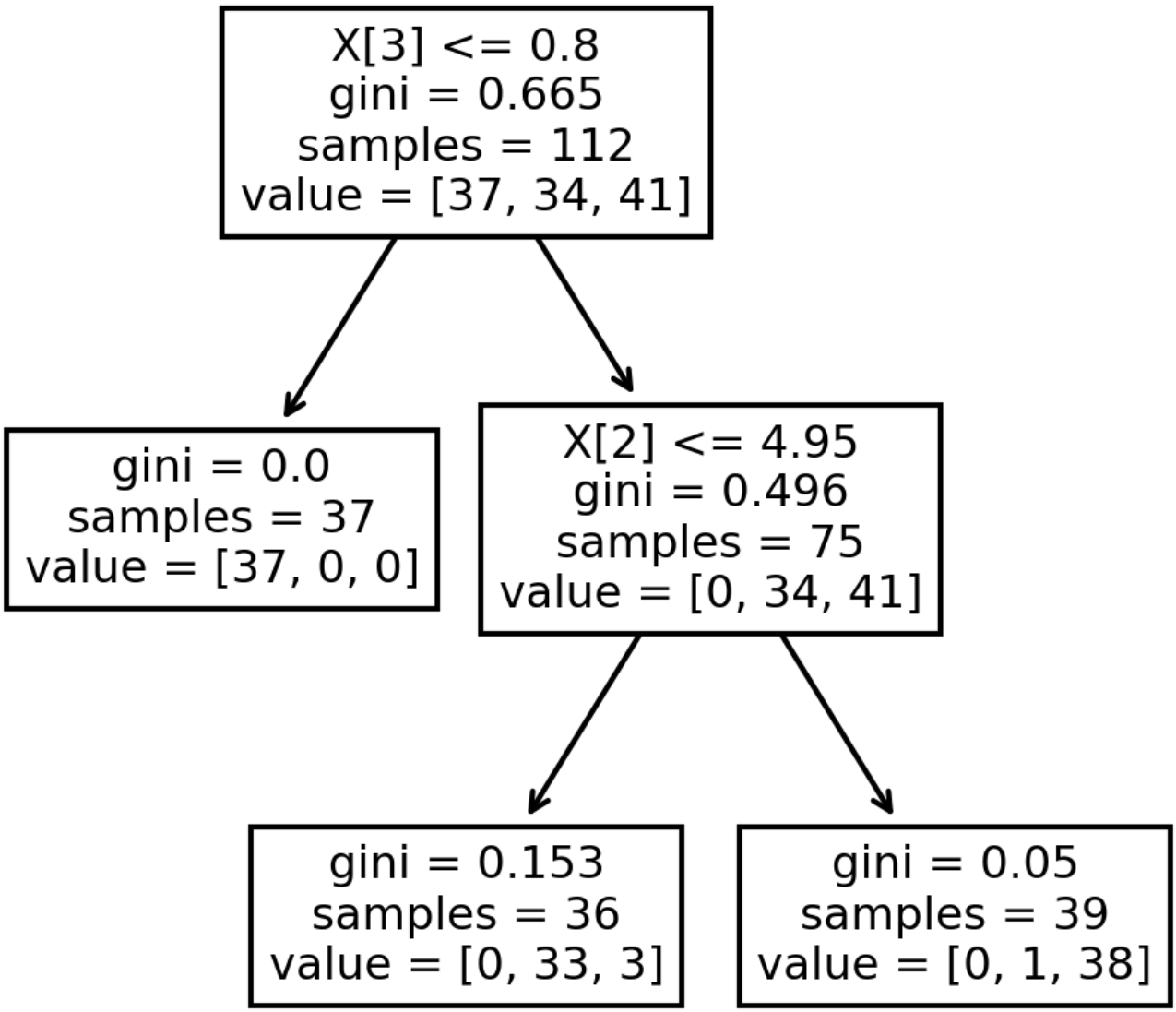
→ Default Visualization Based on the Model

```
tree.plot_tree(clf);
```



→ Adjust Figure Size and Dots per inch (DPI)

```
fig, axes = plt.subplots(nrows = 1, ncols = 1, figsize = (4,4), dpi = 300)

tree.plot_tree(clf);
```

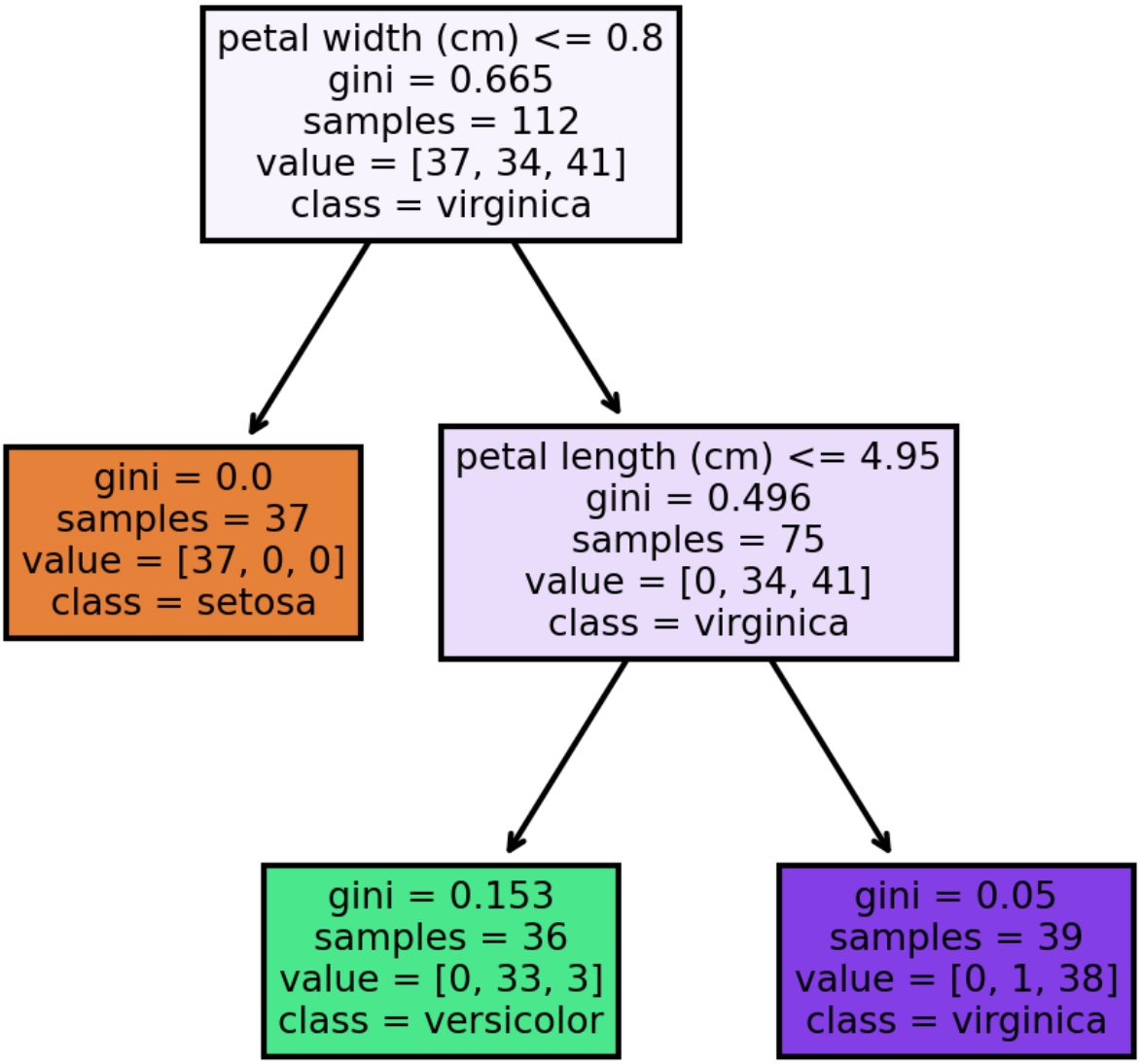


→ Make Tree More Interpretable

The code below not only allows you to save a visualization based on your model, but also makes the decision tree **more interpretable by adding in feature and class names**.

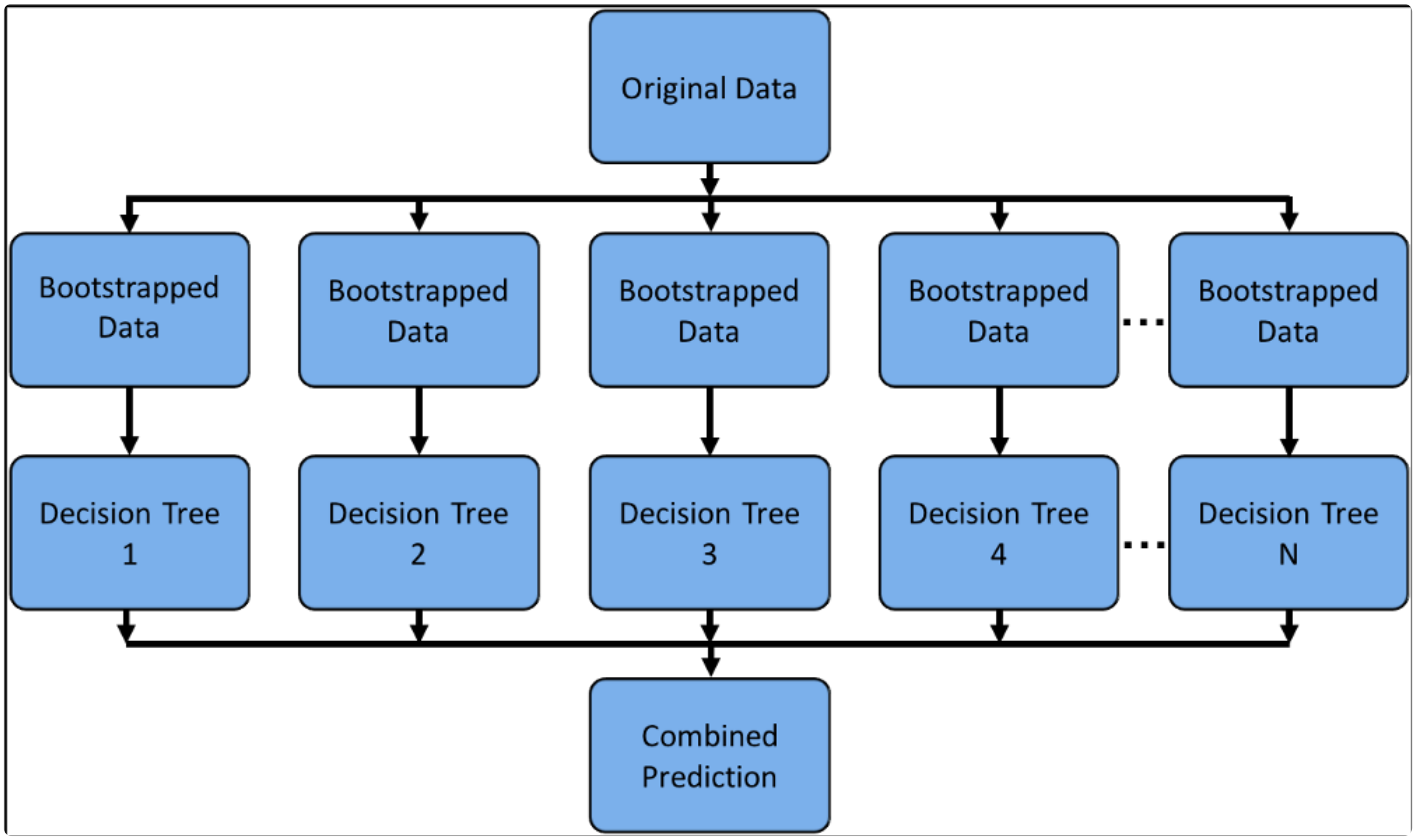
```
# Putting the feature names and class names into variables
fn = ['sepal length (cm)','sepal width (cm)','petal length (cm)','petal width (cm)']
cn = ['setosa', 'versicolor', 'virginica']
```

```
fig, axes = plt.subplots(nrows = 1, ncols = 1, figsize = (4,4), dpi = 300)
tree.plot_tree(clf,
               feature_names = fn,
               class_names=cn,
               filled = True);
fig.savefig('images/plottreefncn.png')
```



⇨ Bagged Trees:

Each machine learning algorithm has strengths and weaknesses. A weakness of decision trees is that they are prone to overfitting on the training set. A way to mitigate this problem is to constrain how large a tree can grow. Bagged trees try to overcome this weakness by using bootstrapped data to grow multiple deep decision trees. The idea is that many trees protect each other from individual weaknesses.



In this video, I'll share with you how you can build a bagged tree model for regression.

Load the Dataset: This dataset contains house sale prices for King County, which includes Seattle. It includes homes sold between May 2014 and May 2015. The code below loads the dataset. The goal of this dataset is to predict price based on features like number of bedrooms and bathrooms

```
df = pd.read_csv('https://www.evanmarie.com/content/files/class_files/scikit-learn-lile

df.head()
```

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	..
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	..
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	..
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	..
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	0	..
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	0	..

5 rows × 21 columns

```
# This notebook only selects a couple features for simplicity
# However, I encourage you to play with adding and substracting more features
features = ['bedrooms','bathrooms','sqft_living','sqft_lot','floors']

X = df.loc[:, features]          # Features Matriz

y = df.loc[:, 'price'].values    # Target Vector
```

→ Splitting Data into Training and Test Sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Note, another benefit of bagged trees like decision trees is that you don't have to standardize your features unlike other algorithms like logistic regression and K-Nearest Neighbors.

→ Bagged Trees

Step 1: Import the model you want to use

In sklearn, all machine learning models are implemented as Python classes

Step 2: Make an instance of the Model

This is a place where we can tune the hyperparameters of a model.

```
reg = BaggingRegressor(n_estimators=100,
                       random_state = 0)
```

Step 3: Training the model on the data, storing the information learned from the data

Model is learning the relationship between X (features like number of bedrooms) and y (price)

```
reg.fit(X_train, y_train)
```

```
BaggingRegressor(n_estimators=100, random_state=0)
```

Step 4: Make Predictions

Uses the information the model learned during the model training process

```
# Returns a NumPy Array
# Predict for One Observation
reg.predict(X_test.iloc[0].values.reshape(1, -1))
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/base.py:451: UserWarning: X does not
have valid feature names, but BaggingRegressor was fitted with feature names
  "X does not have valid feature names, but"
```

```
array([353334.6])
```

Predict for Multiple Observations at Once

```
reg.predict(X_test[0:10])
```

```
array([ 353334.6 , 1011004.77,  450212.76,  418593.  ,  772871.7 ,
        405436.5 ,  361353.02,  720323.9 ,  580438.82, 1623570.8 ])
```

→ Measuring Model Performance

Unlike classification models where a common metric is accuracy, regression models use other metrics like R^2, the coefficient of determination to quantify your model's performance. The best possible score is 1.0. A constant

model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

```
score = reg.score(X_test, y_test)
print(score)
```

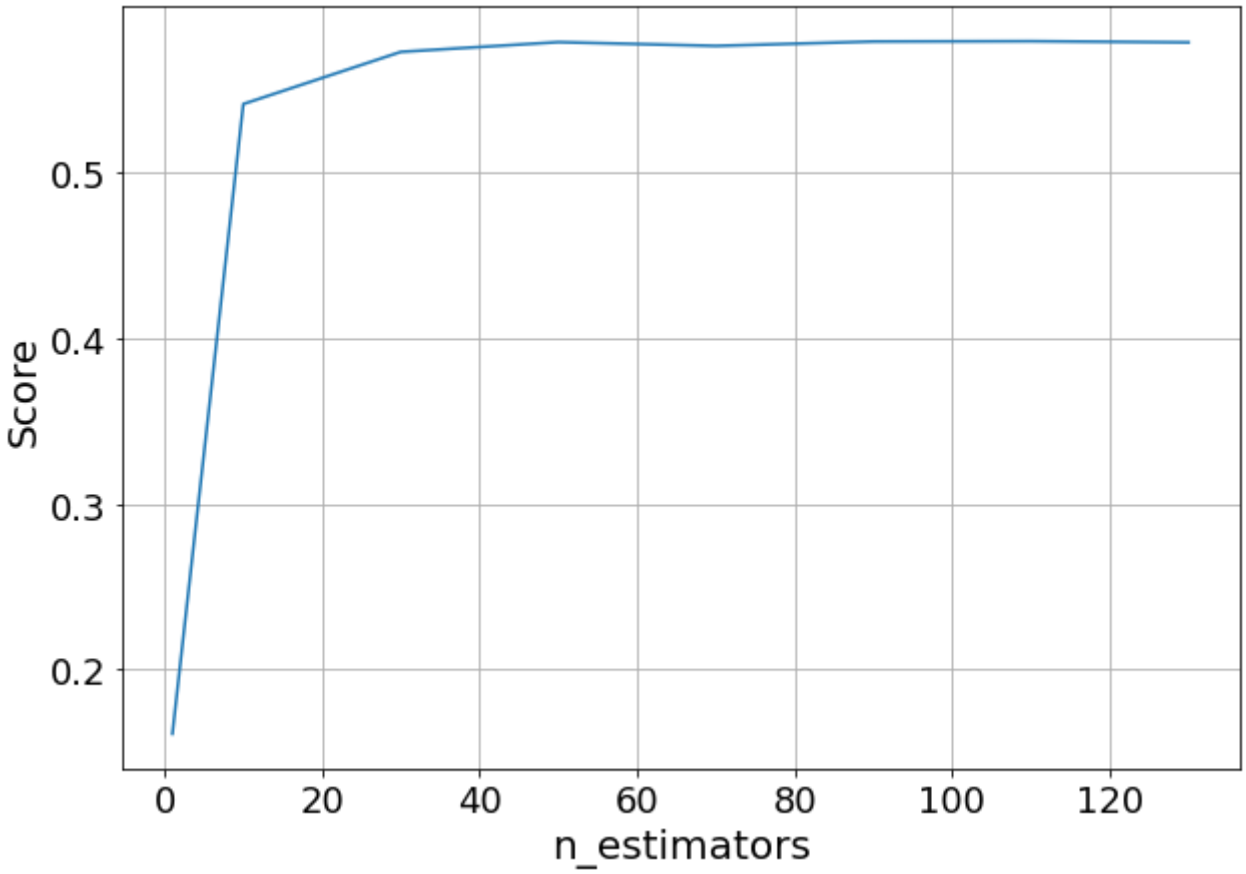
```
0.5786196798753096
```

→ Tuning n_estimators (Number of Decision Trees)

A tuning parameter for bagged trees is **n_estimators**, which represents the number of trees that should be grown.

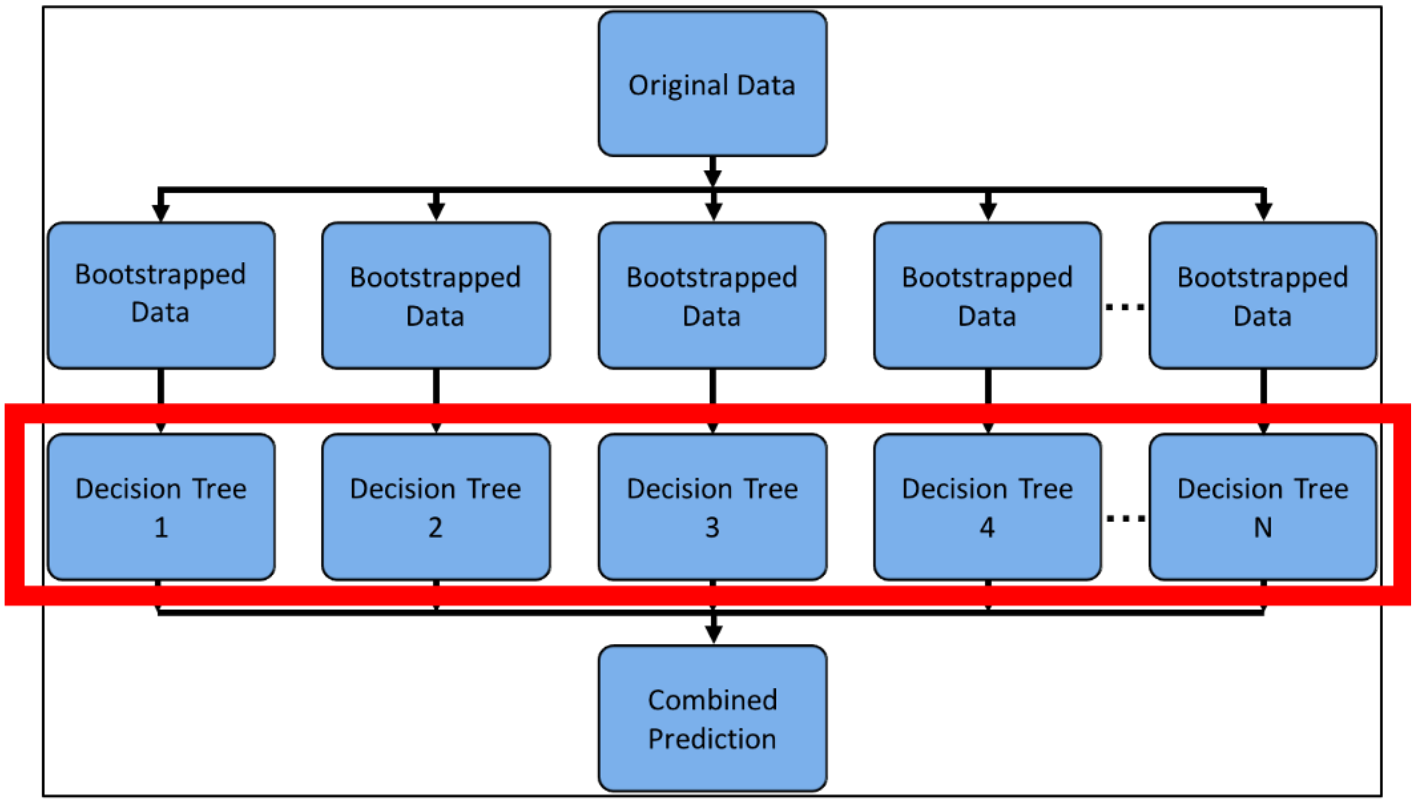
```
# List of values to try for n_estimators:
estimator_range = [1] + list(range(10, 150, 20))

scores = []

for estimator in estimator_range:
    reg = BaggingRegressor(n_estimators=estimator, random_state=0)
    reg.fit(X_train, y_train)
    scores.append(reg.score(X_test, y_test))
```

```
plt.figure(figsize = (10,7))
plt.plot(estimator_range, scores);

plt.xlabel('n_estimators', fontsize =20);
plt.ylabel('Score', fontsize = 20);
plt.tick_params(labelsize = 18)
plt.grid()
```



Notice that the score stops improving after a certain number of estimators (decision trees). One way to get a better score would be to include more features in the features matrix. So that's it, I encourage you to try a building a bagged tree model

Each machine learning algorithm has strengths and weaknesses. Bagged tree models use many trees to protect individual decision trees from overfitting. However, bagged tree models are not without weaknesses. Suppose you have one very strong feature in a dataset, most of the trees will use that feature as the top split. This will result in many similar trees. You can think of Random forests as a variant of a bagged tree model. The difference is that each time a split is considered, only a portion of the total number of features are split candidates. In short, Random Forests make the decision trees less correlated.



The area enclosed in the red rectangle is a place where the random forest algorithm differs from the bagged tree algorithm.

In this video, I'll share with you how you can build a random forest model using Scikit-Learn.

Load the Dataset: This dataset contains house sale prices for King County, which includes Seattle. It includes homes sold between May 2014 and May 2015. The code below loads the dataset. The goal of this dataset is to predict price based on features like number of bedrooms and bathrooms

```
df = pd.read_csv('https://www.evanmarie.com/content/files/class_files/scikit-learn-lile

df.head()
```

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	..
0	7129300520	20141013T000000	221900.0	3	1.00	1180	5650	1.0	0	0	..

	id	date	price	bedrooms	bathrooms	sqft_living	sqft_lot	floors	waterfront	view	..
1	6414100192	20141209T000000	538000.0	3	2.25	2570	7242	2.0	0	0	..
2	5631500400	20150225T000000	180000.0	2	1.00	770	10000	1.0	0	0	..
3	2487200875	20141209T000000	604000.0	4	3.00	1960	5000	1.0	0	0	..
4	1954400510	20150218T000000	510000.0	3	2.00	1680	8080	1.0	0	0	..

5 rows × 21 columns

```
# This notebook only selects a couple features for simplicity
# However, I encourage you to play with adding and substracting features
features = ['bedrooms','bathrooms','sqft_living','sqft_lot','floors']

X = df.loc[:, features]

y = df.loc[:, 'price'].values
```

→ Splitting Data into Training and Test Sets

```
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)
```

Note, another benefit of bagged trees like decision trees is that you don't have to standardize your features unlike other algorithms like logistic regression and K-Nearest Neighbors.

⇨ Random Forest

Step 1: Import the model you want to use

In sklearn, all machine learning models are implemented as Python classes

Step 2: Make an instance of the Model

This is a place where we can tune the hyperparameters of a model.

```
reg = RandomForestRegressor(n_estimators=100, random_state = 0)
```

Step 3: Training the model on the data, storing the information learned from the data

Model is learning the relationship between X (features like number of bedrooms) and y (price)

```
reg.fit(X_train, y_train)
```

```
RandomForestRegressor(random_state=0)
```

Step 4: Predict the labels of new data (new flowers)

Uses the information the model learned during the model training process

```
# Returns a NumPy Array
# Predict for One Observation
```

```
reg.predict(X_test.iloc[0].values.reshape(1, -1))
```

```
/usr/local/lib/python3.7/dist-packages/sklearn/base.py:451: UserWarning: X does not
have valid feature names, but RandomForestRegressor was fitted with feature names
  "X does not have valid feature names, but"
```

```
array([354008.56])
```

Predict for Multiple Observations at Once

```
reg.predict(X_test[0:10])
```

```
array([ 354008.56,  999809.  ,  443760.25,  426332.  ,  760570.2 ,
        408775.5 ,  360030.14,  714794.4 ,  585902.14, 1665779.  ])
```

→ Measuring Model Performance

Unlike classification models where a common metric is accuracy, regression models use other metrics like R^2, the coefficient of determination to quantify your model's performance. The best possible score is 1.0. A constant model that always predicts the expected value of y, disregarding the input features, would get a R^2 score of 0.0.

```
score = reg.score(X_test, y_test)
print(score)
```

```
0.577684658845681
```

Visualizing Individual Decision Trees from a Bagged Tree and Random Forest Model

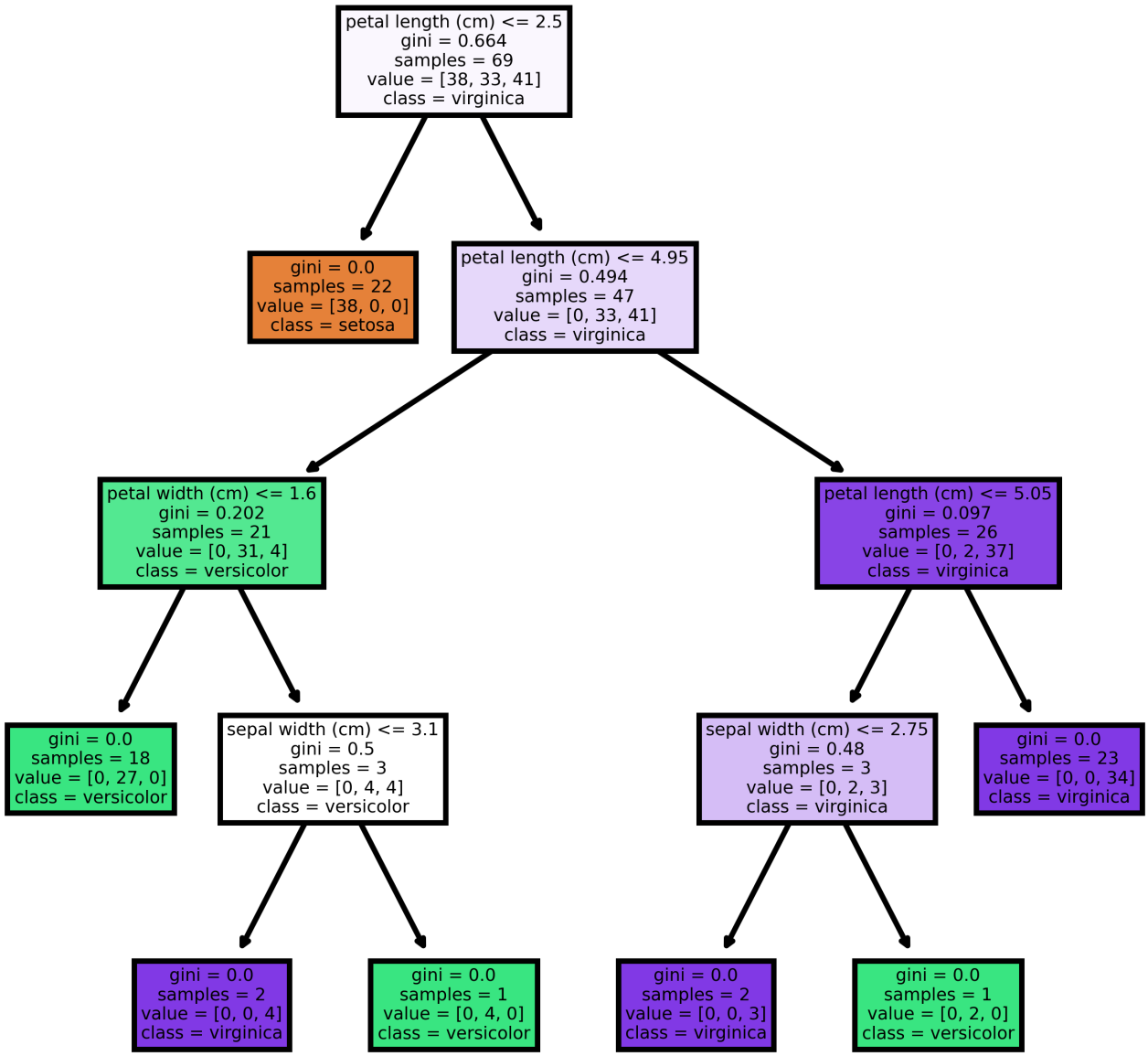
The purpose of this section is to show you that both ensemble models are really comprised of many decision trees.

```
# Load the Iris Dataset
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target

# Split data into training and test sets
X_train, X_test, y_train, y_test = train_test_split(df[data.feature_names], df['target'
```

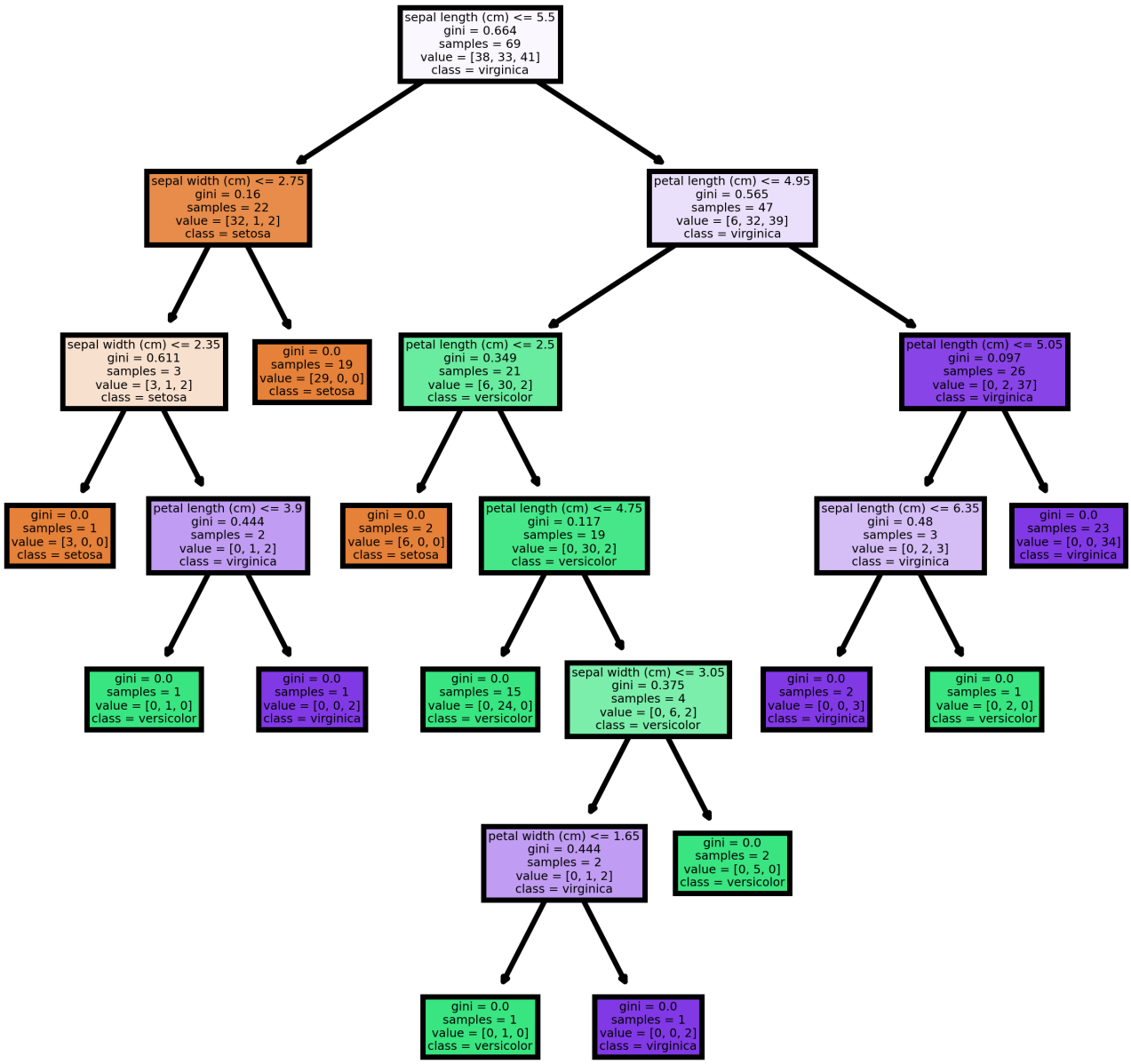
```
# Fir Bagged Tree Model
btc = BaggingClassifier(n_estimators=100,
                        random_state = 1)

btc.fit(X_train, y_train)

# Fit Random Forest Model
rfc = RandomForestClassifier(n_estimators=100, random_state = 1)

rfc.fit(X_train, y_train)
```

```
RandomForestClassifier(random_state=1)
```

```
# Get the first decision tree for bagged tree model
fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (4,4), dpi=800)
tree.plot_tree(btc.estimators_[0],
               feature_names = data.feature_names,
               class_names=data.target_names,
               filled = True);
```




```
# Get the first decision tree for a random forest model
fig, axes = plt.subplots(nrows = 1,ncols = 1,figsize = (4,4), dpi=800)
tree.plot_tree(rfc.estimators_[0],
               feature_names = data.feature_names,
               class_names=data.target_names,
               filled = True);
```



→ Feature Importance

Random Forests give feature importance metrics. Like any metric, it isn't perfect.

```
importances = pd.DataFrame({'feature':X_train.columns,'importance':np.round(rfc.feature
importances = importances.sort_values('importance',ascending=False)
```

```
importances
```

	feature	importance
2	petal length (cm)	0.453
3	petal width (cm)	0.385
0	sepal length (cm)	0.139
1	sepal width (cm)	0.023

Machine Learning with Scikit-learn

Part Three: Unsupervised Learning

Michael Galarnyk, LinkedIn Learning

⇨ Unsupervised Learning:

Unsupervised learning is when you train an algorithm without giving it the answers for examples in your dataset. In the context of Scikit-learn, this means that you only provide a features matrix when you fit your algorithm.

A features matrix is a two-dimensional grid of data where rows represent samples and columns represent features. Unlike supervised learning, there's no target factor. It's important to emphasize that unsupervised algorithms don't make predictions from the data. There are two common types of unsupervised learning algorithms.

The first is clustering. **Clustering** is often used to discover natural groupings in a dataset, when common use is for market segmentation. Companies often have large amounts of customer information. By clustering customers into different segments, they can more efficiently sell or market to their customers.

Another common type of unsupervised learning is **dimensionality reduction**. You can think of dimensionality reduction techniques as data compression algorithms. They can make your data take up less space on your computer. Having fewer features in your data can make visualizing your data easier as well speed up the fitting of your machine learning algorithms. Unsupervised learning helps you discover structure in your data.

⇨ K-Means

Clustering algorithms help identify distinct groups of data. An example is to use clustering to group customers based on their behavior. There are many clustering algorithms, but the most commonly used algorithm is K-Means. In this video, I'll show you how to use K-Means clustering to find some underlying structure in your data.

→ Import Libraries

```
%matplotlib inline

import matplotlib.pyplot as plt
import pandas as pd
import numpy as np

from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

from sklearn.cluster import KMeans
from sklearn.decomposition import PCA

from sklearn.linear_model import LogisticRegression
from sklearn.pipeline import Pipeline
```

Load the Dataset: The Iris dataset is one of datasets scikit-learn comes with that do not require the downloading of any file from some external website. The code below loads the iris dataset.

```
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)
0	5.1	3.5	1.4	0.2
1	4.9	3.0	1.4	0.2
2	4.7	3.2	1.3	0.2
3	4.6	3.1	1.5	0.2
4	5.0	3.6	1.4	0.2

Arrange Data into Features Matrix

K-Means is considered an unsupervised learning algorthm. This means you only need a features matrix. In the iris dataset, there are four features. In this notebook, the features matrix will only be two features as it is easier to visualize clusters in two dimensions.

```
features = ['petal length (cm)','petal width (cm)']

# Create features matrix
x = df.loc[:, features].values
```

The variable y below is for demonstational purposes in this notebook and not needed if you want to do K-Means.

```
y = data.target
```

→ Standardize the Data

KMeans is effected by scale so you need to scale the features in the data before using KMeans. You can transform the data onto unit scale (mean = 0 and variance = 1) for better performance. Scikit-Learn's `StandardScaler` helps standardize the dataset's features.

```
# Apply Standardization to features matrix X
x = df.loc[:, features].values
```

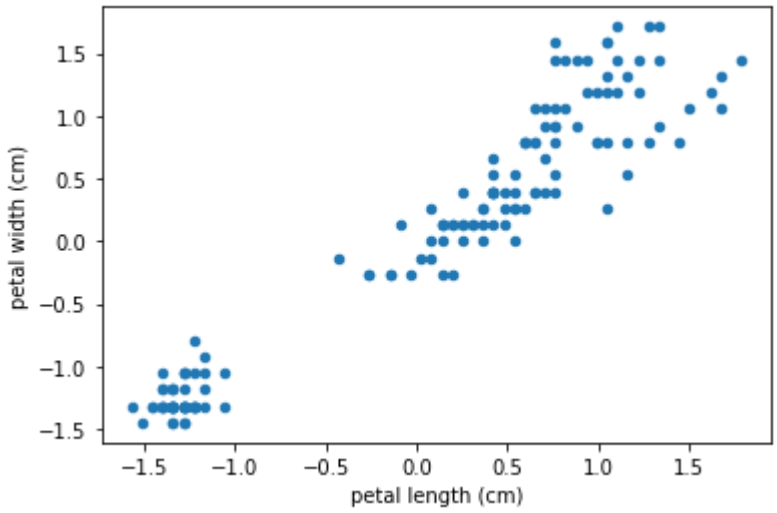
```
x = StandardScaler().fit_transform(x)
```

→ Plot Data to Estimate Number of Clusters

If your data is 2 or three dimensional, it is a good idea to plot your data before clustering. Hopefully you can see if there are any natural looking clusters.

```
# Plot
pd.DataFrame(x, columns = features).plot.scatter('petal length (cm)','petal width (cm)'
```

```
# Add labels
plt.xlabel('petal length (cm)');
plt.ylabel('petal width (cm)');
```



→ KMeans Clustering

In K-Means clustering, you need to specify the number of clusters (k) you want. In Scikit-Learn, this parameter is called `n_clusters` . In the case of the code below, the number of clusters is set to 3 because most people who use the Iris dataset happen to know there are three species.

```
# Make an instance of KMeans with 3 clusters
kmeans = KMeans(n_clusters=3, random_state=1)

# Fit only on a features matrix
kmeans.fit(x)
```

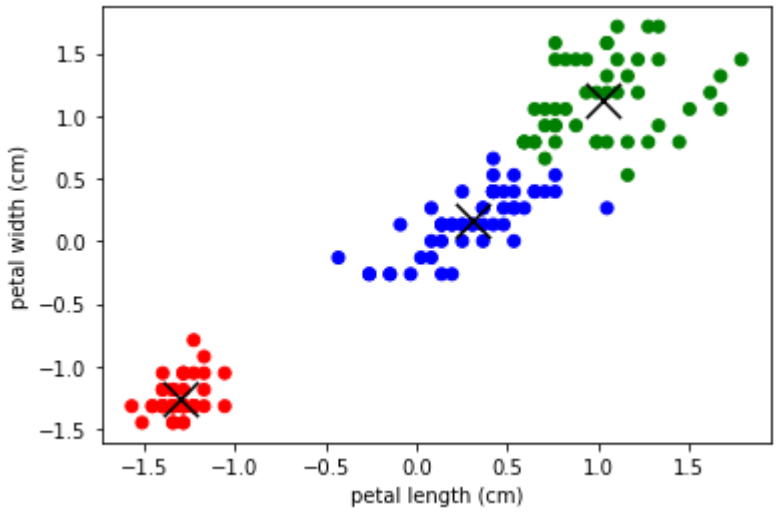
```
KMeans(n_clusters=3, random_state=1)
```

```
# Get labels and cluster centroids
labels = kmeans.labels_
centroids = kmeans.cluster_centers_
```

→ Visually Evaluate the Clusters

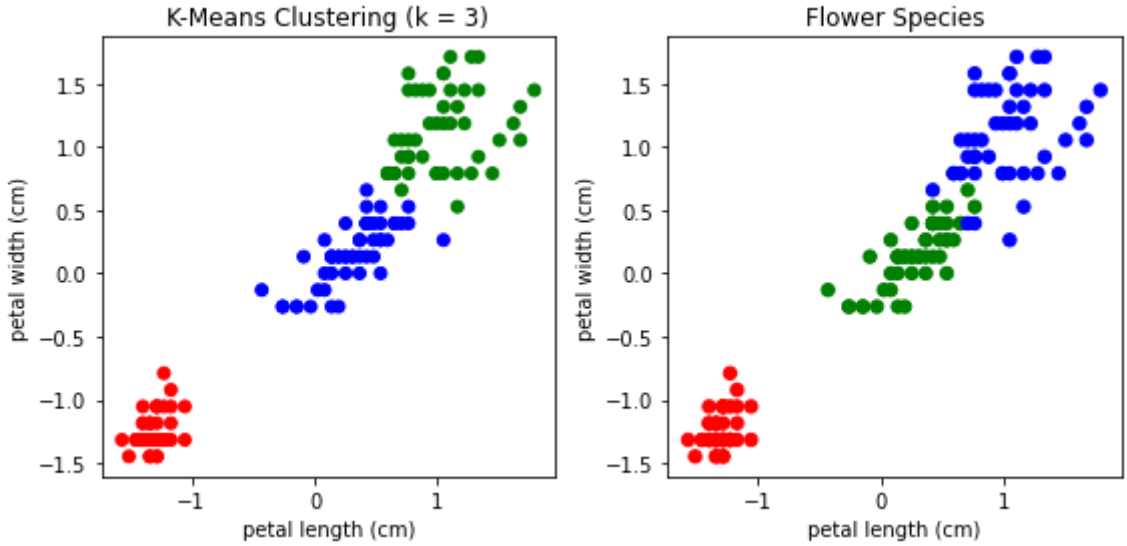
```
x = pd.DataFrame(x, columns = features)
```

```
colormap = np.array(['r', 'g', 'b'])
plt.scatter(x['petal length (cm)'], x['petal width (cm)'], c=colormap[labels])
plt.scatter(centroids[:,0], centroids[:,1], s = 300, marker = 'x', c = 'k')

plt.xlabel('petal length (cm)')
plt.ylabel('petal width (cm)');
```



→ Visually Evaluate the Clusters and Compare Species

```
plt.figure(figsize=(8,4))

plt.subplot(1, 2, 1)
plt.scatter(x['petal length (cm)'], x['petal width (cm)'], c=colormap[labels])
plt.xlabel('petal length (cm)')
plt.ylabel('petal width (cm)');
plt.title('K-Means Clustering (k = 3)')

plt.subplot(1, 2, 2)
plt.scatter(x['petal length (cm)'], x['petal width (cm)'], c=colormap[y], s=40)
plt.xlabel('petal length (cm)')
plt.ylabel('petal width (cm)');
plt.title('Flower Species')

plt.tight_layout()
```



They look pretty similar. Looks like KMeans picked up flower differences with only two features and not the labels. The colors are different in the two graphs simply because KMeans gives out a arbitrary cluster number and the iris dataset has an arbitrary number in the target column.

⇨ PCA for Data Visualization

Are all the features in your dataset needed? Say a you have some flowers and you measured their petal length. If you have a column of that measurement in centimeters and another column with the measurement in inches, do you need both columns? In that circumstance, you can probably drop either column without losing information. In other cases, dropping a column could lead to issues. **Principal component analysis, better known as PCA**, is a technique that you can use to smartly reduce the dimensionality of your dataset while losing the least amount of information possible. One use of PCA is for data visualization. In this video, I'll share with you how you can use PCA to help visualize your data.

Load the Dataset: The Iris dataset is one of datasets scikit-learn comes with that do not require the downloading of any file from some external website. The code below loads the iris dataset.

```
data = load_iris()
df = pd.DataFrame(data.data, columns=data.feature_names)
df['target'] = data.target
```

```
speciesDict = {0: 'setosa', 1:'versicolor', 2:'virginica'}

df.loc[:,'target'] = df.loc[:, 'target'].apply(lambda x: speciesDict[x])
```

```
df.head()
```

	sepal length (cm)	sepal width (cm)	petal length (cm)	petal width (cm)	target
0	5.1	3.5	1.4	0.2	setosa
1	4.9	3.0	1.4	0.2	setosa
2	4.7	3.2	1.3	0.2	setosa
3	4.6	3.1	1.5	0.2	setosa
4	5.0	3.6	1.4	0.2	setosa

→ Standardize the Data

PCA is effected by scale so you need to scale the features in the data before using PCA. You can transform the data onto unit scale (mean = 0 and variance = 1) for better performance. Scikit-Learn's `StandardScaler` helps standardize the dataset's features.

```
# Apply Standardization to features matrix X
features = ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)','petal width (
x = df.loc[:, features].values
y = df.loc[:,['target']].values
```

```
x = StandardScaler().fit_transform(x)
```

→ PCA Projection to 2D

The original data has 4 columns (sepal length, sepal width, petal length, and petal width). The code below projects the original data which is 4 dimensional into 2 dimensions. Note that after dimensionality reduction, there usually

isn't a particular meaning assigned to each principal component. The new components are just the two main dimensions of variation.

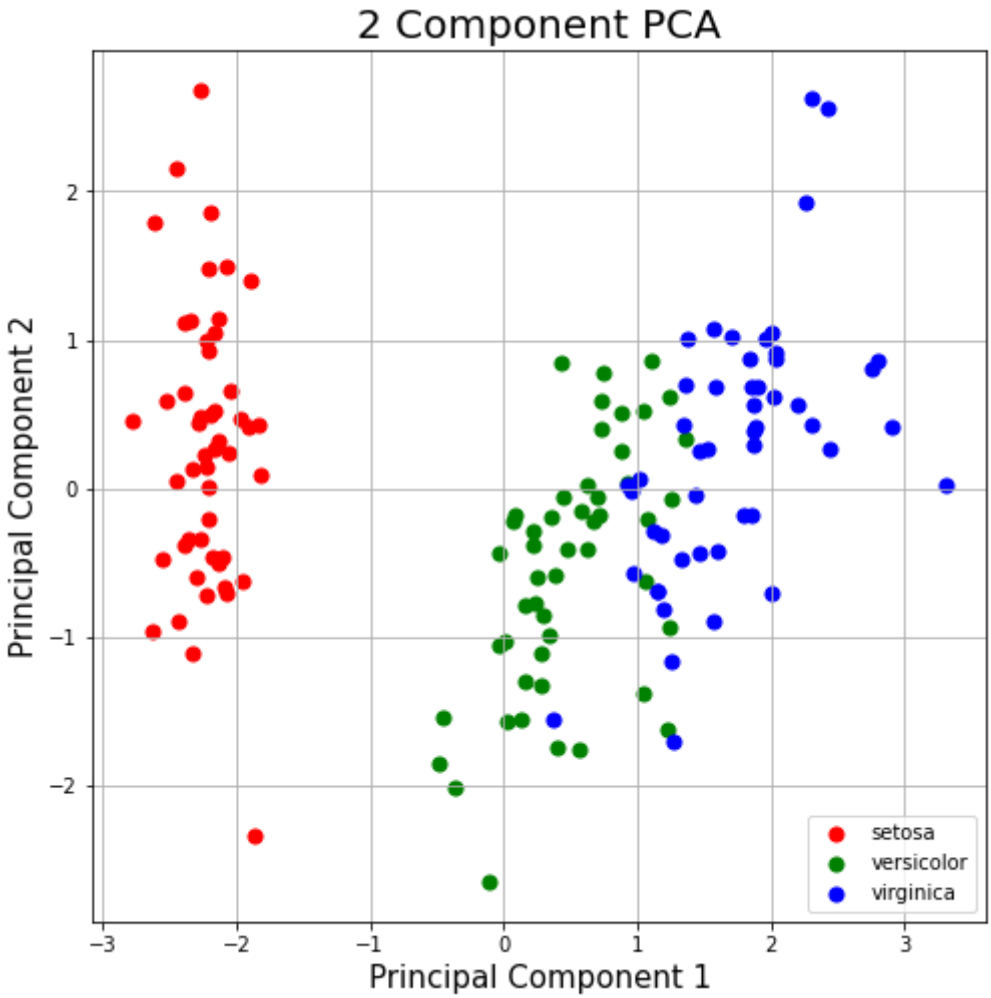
```
# Make an instance of PCA
pca = PCA(n_components=2)

# Fit and transform the data
principalComponents = pca.fit_transform(x)

principalDf = pd.DataFrame(data = principalComponents, columns = ['principal component
```

→ Visualize 2D Projection

PCA projection to 2D to visualize the entire data set.

```
finalDf = pd.concat([principalDf, df[['target']]], axis = 1)
```

```
fig, ax = plt.subplots(nrows = 1, ncols = 1, figsize = (8,8));
targets = df.loc[:, 'target'].unique()
colors = ['r', 'g', 'b']

for target, color in zip(targets,colors):
    indicesToKeep = finalDf['target'] == target
    ax.scatter(finalDf.loc[indicesToKeep, 'principal component 1']
               , finalDf.loc[indicesToKeep, 'principal component 2']
               , c = color
               , s = 50)

ax.set_xlabel('Principal Component 1', fontsize = 15)
ax.set_ylabel('Principal Component 2', fontsize = 15)
ax.set_title('2 Component PCA', fontsize = 20)
ax.legend(targets)
ax.grid()
```



From the graph, it looks like the setosa class is well separated from the versicolor and virginica classes.

→ Explained Variance

The explained variance tells us how much information (variance) can be attributed to each of the principal components. This is important as while you can convert 4 dimensional space to 2 dimensional space, you lose some of the variance (information) when you do this.

```
pca.explained_variance_ratio_
```

```
array([0.72962445, 0.22850762])
```

```
sum(pca.explained_variance_ratio_)
```

```
0.9581320720000165
```

Together, the two principal components contain about 96% of the information. The first principal component contains about 73% of the variance. The second principal component contains about 23% of the variance.

⇨ PCA to Speed Up Machine Learning

Do you want to speed up the fitting of your machine learning algorithm? Scikit-learn offers quite a few ways to do this. One way is to train your model in parallel using the n_jobs parameter which exists for many scikit-learn models. A really simple way is to reduce the number of rows or columns in your data. The problem with this approach is that its hard to know which rows and especially which columns to remove. **Principal component**

analysis, commonly known as PCA, is a technique that you can use to smartly reduce the dimensionality of your dataset while losing the least amount of information possible. In this video, I'll share with you the process of how you can use principal component analysis to speed up the fitting of a logistic regression model.

Load the Dataset: The dataset is a modified version of the MNIST dataset that contains 2000 labeled images of each digit 0 and 1. The images are 28 pixels by 28 pixels.

Parameters	Number
Classes	2 (digits 0 and 1)
Samples per class	2000 samples per class
Samples total	4000
Dimensionality	784 (28 x 28 images)
Features	integers values from 0 to 255

For convenience, I have arranged the data into csv file.

```
df = pd.read_csv('https://www.evanmarie.com/content/files/class_files/scikit-learn-lile
```

```
df.head()
```

	0	1	2	3	4	5	6	7	8	9	...	775	776	777	778	779	780	781	782	783	label
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1

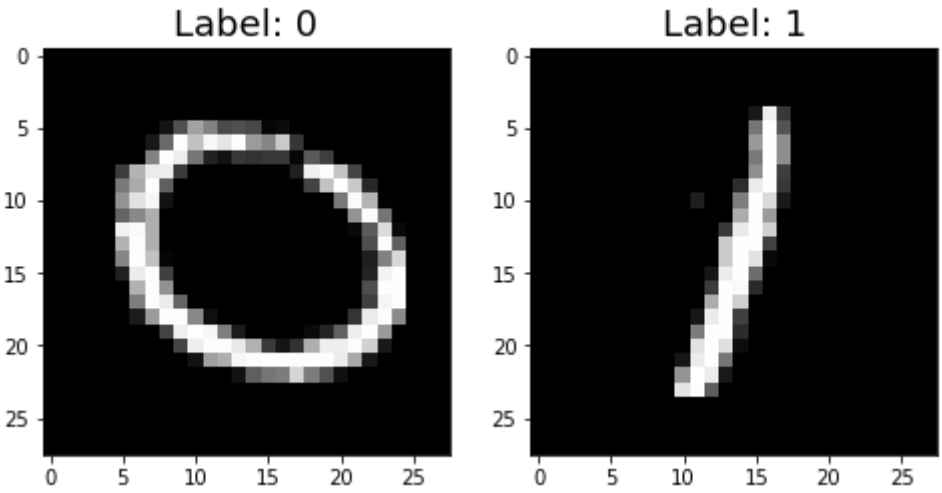
5 rows × 785 columns

→ Visualize Each Digit

```
pixel_colnames = df.columns[:-1]
```

```
# Get all columns except the label column for the first image
image_values = df.loc[0, pixel_colnames].values
```

```
plt.figure(figsize=(8,4))
for index in range(0, 2):

    plt.subplot(1, 2, 1 + index )
    image_values = df.loc[index, pixel_colnames].values
    image_label = df.loc[index, 'label']
    plt.imshow(image_values.reshape(28,28), cmap ='gray')
    plt.title('Label: ' + str(image_label), fontsize = 18)
```



→ Splitting Data into Training and Test Sets

```
X_train, X_test, y_train, y_test = train_test_split(df[pixel_colnames], df['label'], ra
```

→ Standardize the Data

PCA and logisitic regression are sensitive to the scale of your features. You can standardize your data onto unit scale (mean = 0 and variance = 1) by using Scikit-Learn's `StandardScaler` .

```
scaler = StandardScaler()

# Fit on training set only.
scaler.fit(X_train)

# Apply transform to both the training set and the test set.
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

```
# Variable created for demonstational purposes in the notebook
scaledTrainImages = X_train.copy()
```

→ PCA then Logistic Regression

```
"""
n_components = .90 means that scikit-learn will choose the minimum number
of principal components such that 90% of the variance is retained.
"""

pca = PCA(n_components = .90)

# Fit PCA on training set only
pca.fit(X_train)

# Apply the mapping (transform) to both the training set and the test set.
X_train = pca.transform(X_train)
X_test = pca.transform(X_test)
```

```
# Logistic Regression
clf = LogisticRegression()
clf.fit(X_train, y_train)

print('Number of dimensions before PCA: ' + str(len(pixel_colnames)))
print('Number of dimensions after PCA: ' + str(pca.n_components_))
print('Classification accuracy: ' + str(clf.score(X_test, y_test)))
```

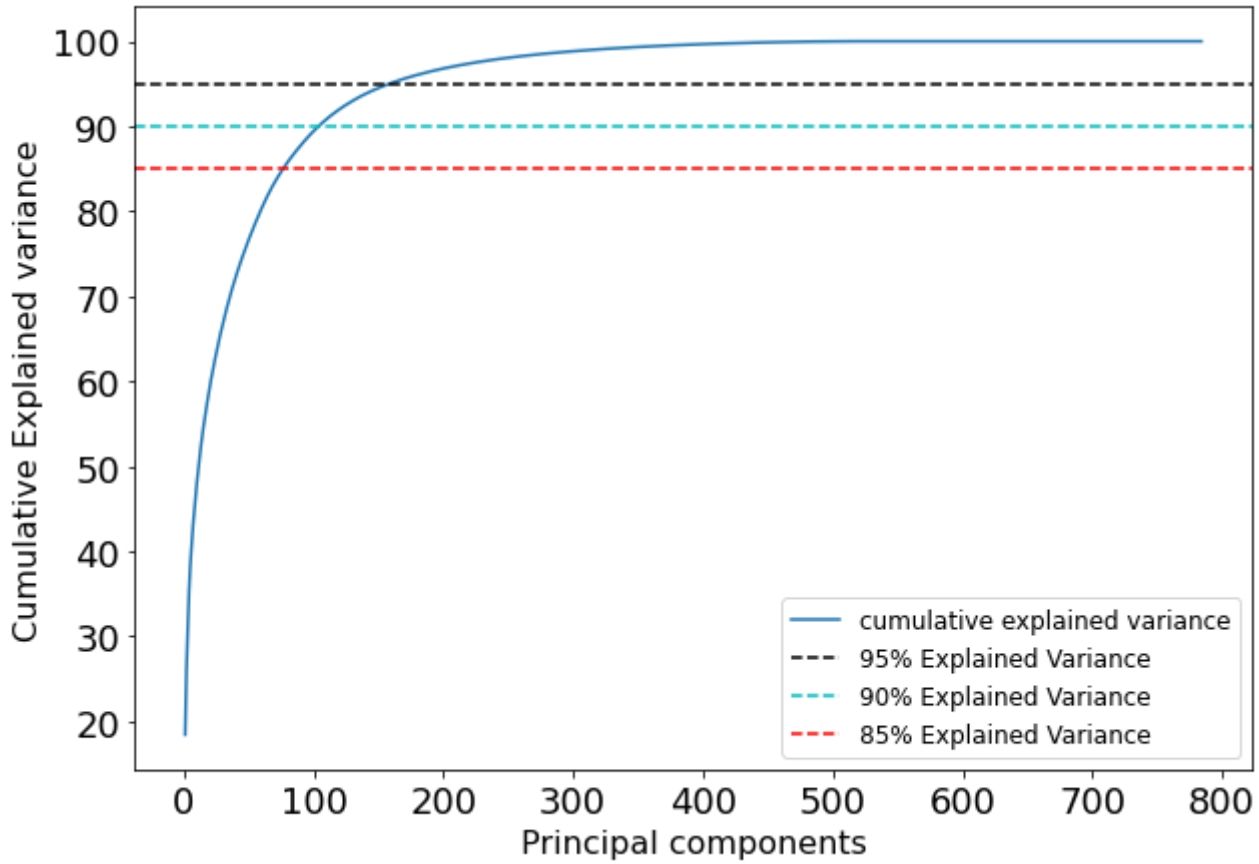
```
Number of dimensions before PCA: 784
Number of dimensions after PCA: 104
Classification accuracy: 0.997
```

→ Relationship between Cumulative Explained Variance and Number of Principal Components

Don't worry if you don't understand the code in this section. It is to show the level of redundancy present in multiple dimensions.

```
# if n_components is not set, all components are kept (784 in this case)
pca = PCA()

pca.fit(scaledTrainImages)

# Summing explained variance
tot = sum(pca.explained_variance_)

var_exp = [(i/tot)*100 for i in sorted(pca.explained_variance_, reverse=True)]

# Cumulative explained variance
cum_var_exp = np.cumsum(var_exp)

# PLOT OUT THE EXPLAINED VARIANCES SUPERIMPOSED
fig, ax = plt.subplots(nrows = 1, ncols = 1, figsize = (10,7));
ax.tick_params(labelsize = 18)
ax.plot(range(1, 785), cum_var_exp, label='cumulative explained variance')
ax.set_ylabel('Cumulative Explained variance', fontsize = 16)
ax.set_xlabel('Principal components', fontsize = 16)
ax.axhline(y = 95, color='k', linestyle='--', label = '95% Explained Variance')
ax.axhline(y = 90, color='c', linestyle='--', label = '90% Explained Variance')
ax.axhline(y = 85, color='r', linestyle='--', label = '85% Explained Variance')
ax.legend(loc='best', markerscale = 1.0, fontsize = 12)
```

```
<matplotlib.legend.Legend at 0x7efdc28c4e90>
```



⇨ Pipelines

Machine learning is not always about applying a single machine learning algorithm. For a lot of machine learning applications, you will need to apply various data processing steps, data transformations, and potentially multiple machine learning algorithms. This can lead to a lot of code. The question becomes, how do you keep your code organized and as bug free as possible? In this video, I'll share with you how can use pipelines in scikit-learn to make your code cleaner and more resilient to bugs.

To demonstrate the utility of pipelines, this notebook shows how much less code you need to chain together pca and logistic regression for image classification.

Load the Dataset: The dataset is a modified version of the MNIST dataset that contains 2000 labeled images of each digit 0 and 1. The images are 28 pixels by 28 pixels.

Parameters	Number
Classes	2 (digits 0 and 1)
Samples per class	2000 samples per class
Samples total	4000
Dimensionality	784 (28 x 28 images)
Features	integers values from 0 to 255

For convenience, I have arranged the data into csv file.

```
df = pd.read_csv('https://www.evanmarie.com/content/files/class_files/scikit-learn-lile
```

```
df.head()
```

	0	1	2	3	4	5	6	7	8	9	...	775	776	777	778	779	780	781	782	783	label
0	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1
2	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	0	0	0	0	0	0	...	0	0	0	0	0	0	0	0	0	1

5 rows × 785 columns

→ Without Pipeline

Notice how many steps this takes. There are quite a few places where an error could occur.

```
# Train Test Split
X_train, X_test, y_train, y_test = train_test_split(df[df.columns[:-1]], df['label'], r

# Standardize Data
scaler = StandardScaler()
scaler.fit(X_train)
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)

# Apply PCA
pca = PCA(n_components = .90, random_state=0)
pca.fit(X_train)
X_train = pca.transform(X_train)
X_test = pca.transform(X_test)

# Apply Logistic Regression
clf = LogisticRegression()
clf.fit(X_train, y_train)

# Get Model Performance
print(clf.score(X_test, y_test))
```

```
0.997
```

→ With Pipelines

```
# Train Test Split
X_train, X_test, y_train, y_test = train_test_split(df[df.columns[:-1]], df['label'], r

# Create a pipeline
pipe = Pipeline([('scaler', StandardScaler()),
                 ('pca', PCA(n_components = .90, random_state=0)),
                 ('logistic', LogisticRegression())])

pipe.fit(X_train, y_train)
```

```
# Get Model Performance
print(pipe.score(X_test, y_test))
```

```
0.997
```

→ Visualize Pipeline

```
from sklearn import set_config

set_config(display='diagram')
pipe
```

```
Pipeline(steps=[('scaler', StandardScaler()),
('pca', PCA(n_components=0.9, random_state=0)),
('logistic', LogisticRegression())])
```

Please rerun this cell to show the HTML repr or trust the notebook.

Pipeline

```
Pipeline(steps=[('scaler', StandardScaler()),
('pca', PCA(n_components=0.9, random_state=0)),
('logistic', LogisticRegression())])
```

StandardScaler

```
StandardScaler()
```

PCA

```
PCA(n_components=0.9, random_state=0)
```

LogisticRegression

```
LogisticRegression()
```