AASD 4000
Machine Learning - I

Applied AI Solutions Developer Program

Module 05
Scikit Learn

Vejey Gandyer

Agenda

- Scikit Learn
- Importance of Scikit Learn
- Setting up scikit-learn
- Modeling Process
- Estimator API
- Use case: Linear Regression problem
- Task 6: Linear Regression model
- Use case: Unsupervised problem

Scikit Learn

What is it?

What is Scikit-learn ?

Scikit-learn (Sklearn) is the most useful and robust library for **machine learning** in Python

It provides a selection of efficient tools for machine learning and statistical modeling including **classification**, **regression**, **clustering** and **dimensionality reduction** via a consistence interface in Python

This library, which is largely written in Python, is built upon **NumPy**, and uses **SciPy** and **Matplotlib**.

Importance of Scikit Learn

Why it is needed?

Importance of Scikit Learn

<u>Faster Prototyping</u>

Amazing features enables the machine learning developer to build models **faster**

Main features of scikit-learn are:

- Dimensionality Reduction
- Regression
- Preprocessing
- Classification
- Model Selection
- Clustering

Setting up Scikit-learn

Installing Scikit-learn

```
>>> import sklearn
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ModuleNotFoundError: No module named 'sklearn'
```

```
>>> import scikit-learn
  File "<stdin>", line 1
    import scikit-learn
                 ^
SyntaxError: invalid syntax
>>> import sklearn
>>>
```

Conda / Miniconda:

conda search scikit-learn

conda install scikit-learn

Developer version

conda install -c anaconda git

pip install Cython

pip install h5py

pip install git+git://github.com/scikit-learn/scikit-learn.git

Virtual environment:

pip install –U scikit-learn

Running Scikit learn

```
In [1]: import scikit-learn
  File "<ipython-input-1-56e12c41cf9d>", line 1
    import scikit-learn
                 ^
SyntaxError: invalid syntax
```

Importing Scikit-learn

Note: sklearn not scikit-learn

```
In [2]: import sklearn

In [3]:
```

Getting Help

```
In [4]: from sklearn.
   base          covariance           discriminant_analysis  externals           impute
   calibration   cross_decomposition  dummy                  feature_extraction  isotonic
   cluster       datasets             ensemble               feature_selection   kernel_approximation  >
   compose       decomposition        exceptions             gaussian_process    kernel_ridge
```

Modeling Process

Modeling Process

Modeling Process involves the following steps:

1. Load the dataset
2. Preprocess the dataset *
3. Split the dataset
4. Train the model
5. Persist the model

* Detailed in a future lecture

Modeling Process

Loading the dataset

Loading the dataset

Dataset: Collection of data

Features: Variables of data aka predictors, inputs or attributes

- Feature matrix: Collection of features
- Feature Names: List of all the names of the features

Response: Output variable depends on feature variables aka target, label or output

- Response Vector: Used to represent response column
- Target Names: Possible values taken by a response vector

Scikit-learn has in-built datasets

- Classification: Iris and digits
- Regression: Boston House Prices

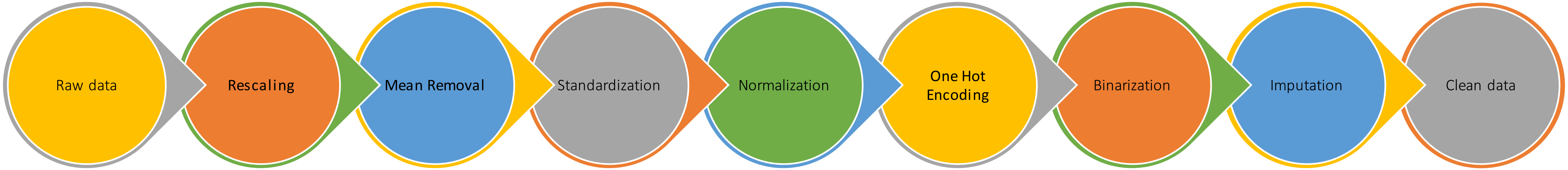
```python
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names
print("Feature names:", feature_names)
print("Target names:", target_names)
print("\nFirst 10 rows of X:\n", X[:10])
```

```
Feature names: ['sepal length (cm)', 'sepal width (cm)',
'petal length (cm)', 'petal width (cm)']
Target names: ['setosa' 'versicolor' 'virginica']
First 10 rows of X:
[[5.1 3.5 1.4 0.2]
[4.9 3.  1.4 0.2]
[4.7 3.2 1.3 0.2]
[4.6 3.1 1.5 0.2]
[5.  3.6 1.4 0.2]
[5.4 3.9 1.7 0.4]
[4.6 3.4 1.4 0.3]
[5.  3.4 1.5 0.2]
[4.4 2.9 1.4 0.2]
[4.9 3.1 1.5 0.1]]
```

Modeling Process

Preprocess the dataset

Data Preprocessing



To be seen in detail in a future lecture

Modeling Process

Split the dataset

Splitting the dataset

Training Set: Used to train the model

Testing Set: Used to test the model and not used for training

Split: Ratio of Training Set to Testing Set (70:30 / 80:20)

```python
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.3,
                                                    random_state=1
)
print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(105, 4)
(45, 4)
(105,)
(45,)
```

- X
- y
- test_size
- random_state

Modeling Process

Train the model

Train the model

Model to be trained using the training set

Choose a model to use

Build an object for the model with specific parameters

Fit the model

Predict for the new unseen data

```python
# Load dataset
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target

# Split dataset
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.4,
random_state=1)

from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

# Build the model object
classifier_knn = KNeighborsClassifier(n_neighbors=3)

# Fit the model
classifier_knn.fit(X_train, y_train)

# Predict on testing dataset
y_pred = classifier_knn.predict(X_test)

# Finding accuracy by comparing actual response values(y_test)with predicted
response value(y_pred)
print("Accuracy:", metrics.accuracy_score(y_test, y_pred))

# Providing sample data and the model will make prediction out of that data
sample = [[5, 5, 3, 2], [2, 4, 3, 5]]
preds = classifier_knn.predict(sample)
pred_species = [iris.target_names[p] for p in preds]
print("Predictions:", pred_species)
```

Model - kNN

fit()

predict()

```
Accuracy: 0.9833333333333333
Predictions: ['versicolor', 'virginica']
```

Modeling Process

Persist the model

Persist the model

Trained Model must be stored and persisted for future use

joblib package

dump() - save the trained model

load() - load the stored model for use

```python
# Saving the trained model using dump()
from sklearn.externals import joblib
joblib.dump(classifier_knn, 'iris_classifier_knn.joblib')

# Loading the saved model using load()
joblib.load('iris_classifier_knn.joblib')
```

joblib

dump()

load()

Estimator API

Steps in using Estimator API

1. Choose a class of model
2. Choose model hyperparameters
3. Arranging the data for the model
4. Model Fitting
5. Applying the model

#1 - Choosing a model

Choosing a model is as simple as importing the appropriate Estimator class from Sklearn package

Model: **LinearRegression**

```
from sklearn.linear_model import LinearRegression
```

#2 - Choosing model hyperparameters

Look at all hyperparamaters for a model in its signature

```
LinearRegression(
    fit_intercept=True,
    normalize=False,
    copy_X=True,
    n_jobs=None,
)
```

```
Parameters
----------
fit_intercept : boolean, optional, default True
    whether to calculate the intercept for this model. If set
    to False, no intercept will be used in calculations
    (e.g. data is expected to be already centered).

normalize : boolean, optional, default False
    This parameter is ignored when ``fit_intercept`` is set to False.
    If True, the regressors X will be normalized before regression by
    subtracting the mean and dividing by the l2-norm.
    If you wish to standardize, please use
    :class:`sklearn.preprocessing.StandardScaler` before calling ``fit`` on
    an estimator with ``normalize=False``.

copy_X : boolean, optional, default True
    If True, X will be copied; else, it may be overwritten.

n_jobs : int or None, optional (default=None)
    The number of jobs to use for the computation. This will only provide
    speedup for n_targets > 1 and sufficient large problems.
    ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
    ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
    for more details.
```

Hyperparameter:
fit_intercept

```
model = LinearRegression(fit_intercept=True)
model
```

#3 - Arranging the data

Arrange X, y in its correct dimensions

y - target variable of length n_samples (1D array)

X - feature matrix of size n_samples x n_features

```
X = x[:, np.newaxis]
X.shape
```

Output:

#4 - Fitting the model

Model training happens here

fit()

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

Output:

```
model.coef_
model.intercept_
```

#5 - Applying the model

Apply the model to the new unseen test data

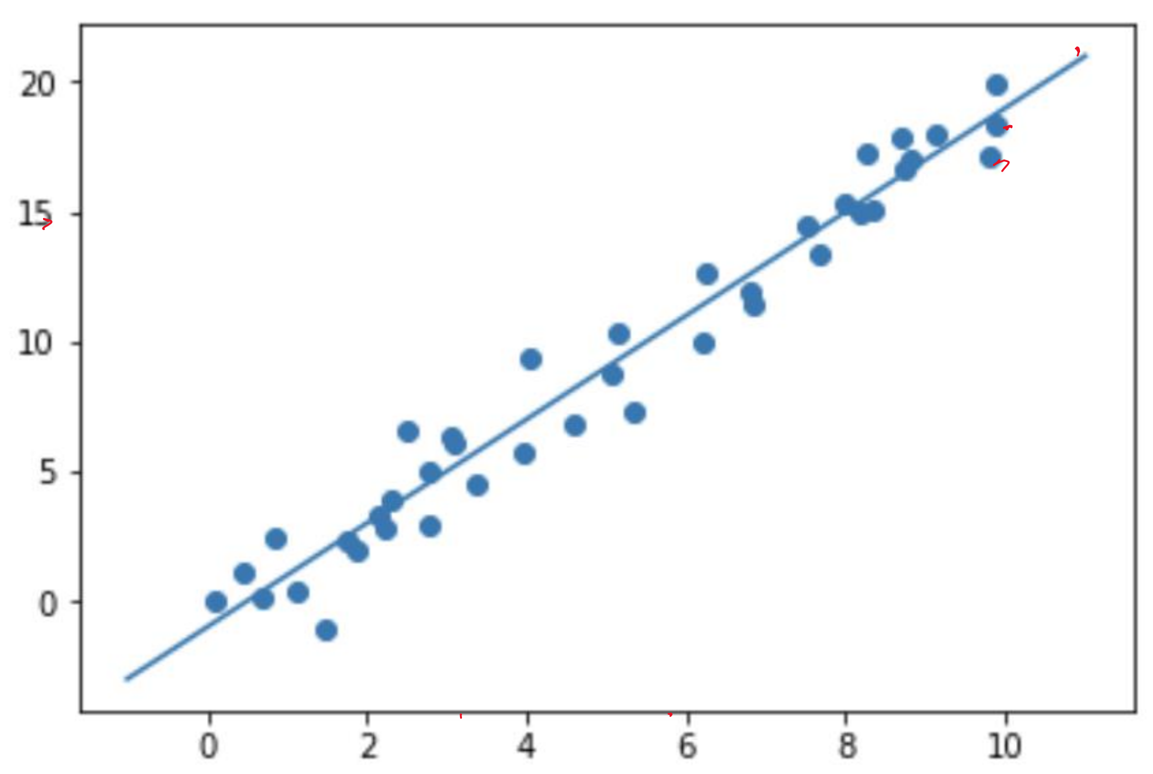
Classification problem: predict()

Unsupervised problem: fit() / transform()

```python
xfit = np.linspace(-1, 11)
Xfit = xfit[:, np.newaxis]
yfit = model.predict(Xfit)
```

Output:

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



Recap: 5 Easy Steps in using Estimator API

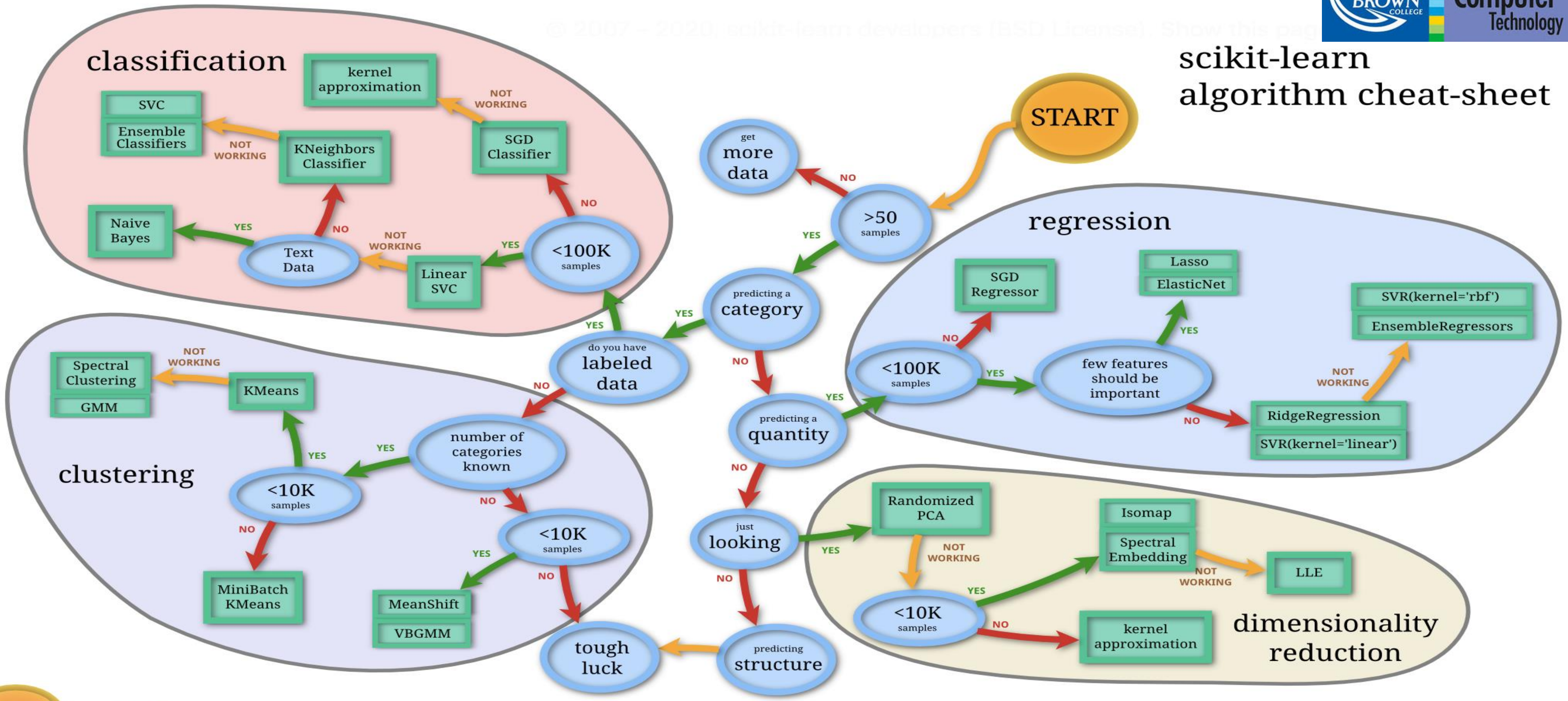
1. Choose a class of model
2. Choose model hyperparameters
3. Arranging the data for the model
4. Model Fitting
5. Applying the model

Estimator API

Scikit-learn Cheatsheet

https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html

scikit-learn algorithm cheat-sheet



Back

Regression Problem

Task 6: Create a Linear Regression model for Real Estate Housing Prices

Task 6: Create a Linear Regression model

Use the previous code scripts in creating a Linear Regression model for the below dataset

Real Estate Housing Prices Dataset

Unsupervised Problem

Steps in using Estimator API

1. Choose a class of model
2. Choose model hyperparameters
3. Model Fitting
4. Transform the data
5. Visualizing the model

Choosing a model

Choosing a model is as simple as importing the appropriate Estimator class from Sklearn package

Model: **PCA** (Principal Component Analysis)

```
from sklearn.decomposition import PCA
```

Choosing model hyperparameters

Look at all hyperparamaters
for a model in its signature

```
PCA(
    n_components=None,
    copy=True,
    whiten=False,
    svd_solver='auto',
    tol=0.0,
    iterated_power='auto',
    random_state=None,
)
```

Hyperparameter: n_components

```
model = PCA(n_components=2)
model
```

```
Parameters
----------
n_components : int, float, None or string
    Number of components to keep.
    if n_components is not set all components are kept::

        n_components == min(n_samples, n_features)

    If ``n_components == 'mle'`` and ``svd_solver == 'full'``, Minka's
    MLE is used to guess the dimension. Use of ``n_components == 'mle'``
    will interpret ``svd_solver == 'auto'`` as ``svd_solver == 'full'``.

    If ``0 < n_components < 1`` and ``svd_solver == 'full'``, select the
    number of components such that the amount of variance that needs to be
    explained is greater than the percentage specified by n_components.

    If ``svd_solver == 'arpack'``, the number of components must be
    strictly less than the minimum of n_features and n_samples.

    Hence, the None case results in::

        n_components == min(n_samples, n_features) - 1

copy : bool (default True)
    If False, data passed to fit are overwritten and running
    fit(X).transform(X) will not yield the expected results,
    use fit_transform(X) instead.

whiten : bool, optional (default False)
    When True (False by default) the `components_` vectors are multiplied
    by the square root of n_samples and then divided by the singular values
    to ensure uncorrelated outputs with unit component-wise variances.

    Whitening will remove some information from the transformed signal
    (the relative variance scales of the components) but can sometime
    improve the predictive accuracy of the downstream estimators by
    making their data respect some hard-wired assumptions.

svd_solver : string {'auto', 'full', 'arpack', 'randomized'}
```

Fitting the model

Model training happens here

fit()

```
model.fit(X_iris)
```

Output:

```
PCA(copy=True, iterated_power='auto', n_components=2, random_state=None,
  svd_solver='auto', tol=0.0, whiten=False)
```

Transform the data

For PCA, no predict()

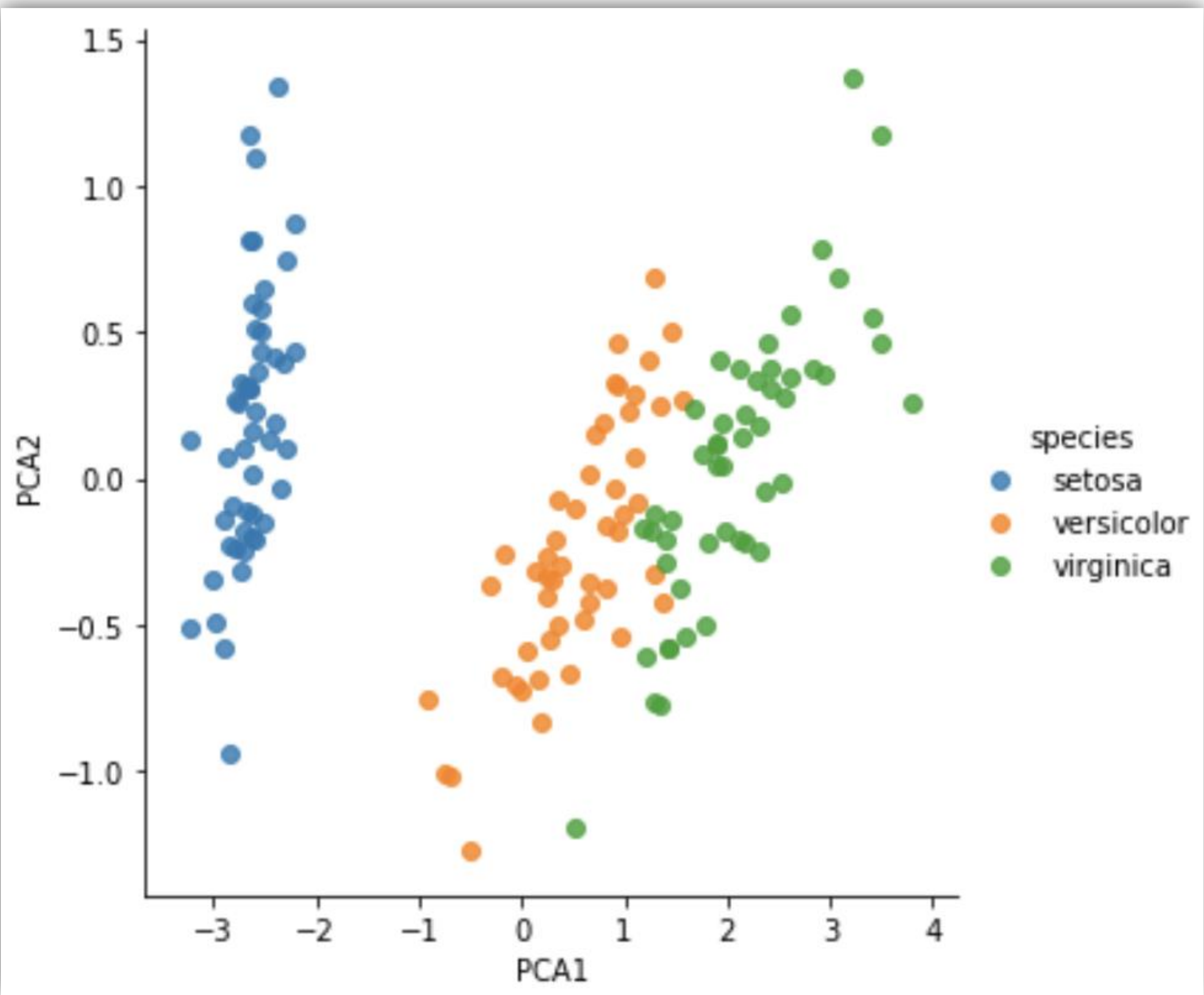
Note transform() method

```
X_2D = model.transform(X_iris)
```

Visualizing the transformed data

```python
X_2D = model.transform(X_iris)
```

```python
iris['PCA1'] = X_2D[:, 0]
iris['PCA2'] = X_2D[:, 1]
sns.lmplot("PCA1", "PCA2",
           hue='species',
           data=iris,
           fit_reg=False
)
```



References & Further Reading

Scikit-learn

https://scikit-learn.org/stable/user_guide.html

Kevin Markham (Data School) on using Scikit-learn

https://www.youtube.com/playlist?list=PL5-da3qGB5ICeMbQuqbbCOQWcS6OYBr5A