# Project 5: Making an AI

## 1. Table of Display

|  |  |
| --- | --- |
| Topic | Method |
| What is ML | Summary |
| Intro to Scikit-Learn | Summary w/Code |
| Hyperparameters and Model Validation | ? |
| Feature Engineering | ? |
| DTrees and Random Forests | ? |
| Face Detection Pipeline | Final Project |

## 2. Table Of Contents

1. [1. Table of Display](#Xc4deb9216b458951b6073418e5fecaebe1f969d)
2. [2. Table Of Contents](#X6b2593602a0c62157c0c7b32214e273425badb5)
3. [3. Background](#X87094d5cc2003181c30bd3fb8c9e78d4fb0c530)
   1. [3.1. What is Machine Learning?](#X81841959dca647c1532650dda7254ec65bada34)
      1. [3.1.1. Supervised learning](#X03412e3cc4c7d0ed5fc93ecd99ad0259824aa66)
         1. [3.1.1.1. Classification](#X31867cb821650aa99643e826dc1d1317f1b26e3)
         2. [3.1.1.2. Regression](#X40c96b31a47f673b789464d8f95b8bededf8505)
      2. [3.1.2. Unspervised Learning](#X4ab5e899edbf9127897d6851e12ed3ae4eb3469)
         1. [3.1.2.1. Clustering](#Xb8b666db3b593d5a59b613a07757ce3946b5297)
         2. [3.1.2.2. Dimensionality Reduction](#X9dbf7eaf319386673b3ad5445d91ef0ed872b1d)
   2. [3.2. Scikit-Learn](#X15531e633f86572a00ffd396cd3017604f047a8)
      1. [3.2.1. Data Structure](#X6f4d17f238ab976166d875be84fec55327a8d23)
      2. [3.2.2. Imputation: Filling Missing Data](#X9b9e5fb895f9faf34ad1c0cb64e717ba4471aca)
   3. [Feature Engineering: What are we working with?](#Xe59ce2d3dda91915fcd6d73487bd378b7957c26)
      1. [Categorical Data](#categorical-data)
      2. [Language Features](#language-features)
      3. [Image Featuers](#image-featuers)
      4. [Derived Features](#derived-features)
   4. [Picking a Model: What will you use?](#picking-a-model-what-will-you-use)
      1. [Hyperparemters](#hyperparemters)
      2. [Validation](#validation)
      3. [Picking The Best Model](#picking-the-best-model)
   5. [Piplines](#piplines)
   6. [3.4. Special Topic: Random Forests](#X2052198495ce7db9bce7fa4ed72384b2c25caf4)
4. [4. Project: Face Detection Pipeline](#X4f41e06d1c7ba6ac291f933ab9e02168210fdc5)
   1. [4.1. Output](#Xbd406c83490d69e1a9a7a43516ca9e7c327b82f)
   2. [4.2. Code](#X0f9131522ceeaa9d542a31395d809401f29d4c6)

## 3. Background

### 3.1. What is Machine Learning?

Machine Learning (ML) is the art of designing mathematical models of data that can be taught via tunable parameters. Because these algorithms are designed to asist in understanding data, there can be some debate of whether ML could be considered a branch of Artificial Intelligence (AI) anymore.

Because ML works with big data which vary greatly in both size, complexity there must also be various methods of analizing this data. The two general categories of ML are **Supervised** and **Unsupervised** learning. We will explorw these methods futher in this paper.

#### 3.1.1. Supervised learning

Supervised learning takes data and labels associated with the data to model their relationship. This ML model is used to apply labels to novel data. Supervised learning is commonly subdivided into **classification** and **regression**.

##### 3.1.1.1. Classification

Classification models use descrite categories such as a status. This type of model may be used to identify objects in an image or seek for when a part might be damaged within a piece of machinery.

This model will require the devleoper to fist make a labeled dataset. Then you must design model inputs and the general assumptions your can provide. After this you can provide a set of model paramters which can be adjusted by the model during the training stage. The end requlst is that when introduced to novel data the classical model will provide a predictive label.

##### 3.1.1.2. Regression

Regression models place their categories in a continuous spectrums. This can be used to seek the pobablity of damage to an piece of machinery or seek more complex relationships such as genetics to regions.

Regression models could be treadted as the oposite of [Demensionality Reduction](#X9dbf7eaf319386673b3ad5445d91ef0ed872b1d) models. They will extract a new unknown relationship and create a new dimesion of labels.

#### 3.1.2. Unspervised Learning

Unsuprvised learning results from not having a labeled dataset. This type of learning is used to find relationships within a data set. Unspervised Learning can be further subdivided into **Clustering** and **Demensitionaly Reduction** models.

##### 3.1.2.1. Clustering

Clustering models act simlarly to classification models exceept they are seeking the distinct groups with the dataset. This can be used to seekout groups within massive datasets that humans may never see.

One method to do clustersing is through the *k*-means fits model. This method finds the center of data clusters, it must be profivided with a value *k* or this value may be tunable. This model seeks the postion of centers that has the minimum distance between all points in the dataset.

##### 3.1.2.2. Dimensionality Reduction

Demensionality Reduction as its name suggests is designed to simpfly a data set into the smalled scturcure possible. This type of model could be used to seek important parameters to watch in larger systems which may take years for a human to analyze. This method allows us to infer new strutures that may not be labels (or may not exist).

A demnstionality reduction model typcially will remove one or more of the layers of data. This could be used to review a large dataset and place the data neatly into graphs that huamns and easily understand, interpret and apply.

### 3.2. Scikit-Learn

While developing a ML algrithem from scratch is possible it could be cumbersom and likly has already been done. Thus to prevent reinventing the wheel, you can use the modeule **Scikit-Learn**. Scikit-Learn contains many of the popular algorithems in one consitent API.

Designed by a group of Cornell University students, Scikit-Learn is intended to be a new user friendly API which can also be useful for experts in the field. To this end Scikit-Learn features consistent usage, inspecettable attributes, simplistic set of classes, and senible defaults for all classes.

#### 3.2.1. Data Structure

Scikit-Learn descibes data with a *features matrix* this matrix can be stored in a NumPy array or Pandas DataFrame. The rows of the features matrix are often called sameles and the columns are considered features. Thus the features matrix is considered to have the shape [n\_samples, n\_features].

The features matrix is then coupled with a *label array*. This array is typically is a Numpy array or Pandas Series of n\_samples, but it can be two-dimesional with the shapre [n\_smaple, n\_targets]. The label array could be considered the dependent variable the model is attempting to predict.

#### 3.2.2. Imputation: Filling Missing Data

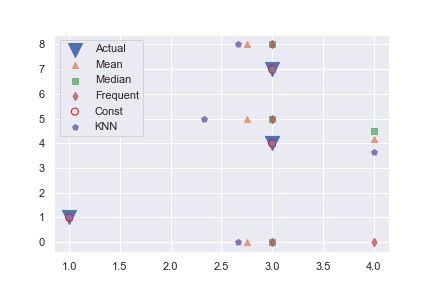
In an ideal world we would have all the data avalible to us perfectly setup and ready to be used. This is rarly the case. To make filling in data smoother sklearn includes the *Imputer* class. This class uses various stregies to fill you datasheets.

To use the Imputer class use:

from sklearn.impute import SimpleImputer

Then you can assign the imputer to a variable and use the .giy\_transform() function. As seen in the example below

### IMPUTATION HOW TO ###  
# %%  
import matplotlib.pyplot as plt  
import numpy as np  
import seaborn as sns  
from numpy import nan  
from sklearn.impute import SimpleImputer  
from sklearn.impute import KNNImputer  
  
data\_main = np.array([[nan, 0, 3],  
 [3, 7, 9],  
 [nan, 5, nan],  
 [4, nan, 6],  
 [nan, 8, 1],  
 [1, 1, 1],  
 [3, 4, 9]])  
imp = SimpleImputer(strategy='mean')  
data\_mean = imp.fit\_transform(data\_main)  
imp = SimpleImputer(strategy='median')  
data\_median = imp.fit\_transform(data\_main)  
imp = SimpleImputer(strategy='most\_frequent')  
data\_freq = imp.fit\_transform(data\_main)  
imp = SimpleImputer(strategy='constant', fill\_value=2)  
data\_const = imp.fit\_transform(data\_main)  
imp = KNNImputer(n\_neighbors=3)  
data\_KNN = imp.fit\_transform(data\_main)  
# %%  
print(data\_main)  
print('--------------')  
print(data\_mean)  
print('--------------')  
print(data\_median)  
print('--------------')  
print(data\_freq)  
print('--------------')  
print(data\_const)  
print('--------------')  
print(data\_KNN)  
  
# %%  
sns.set()  
plt.scatter(data\_main[:, 0], data\_main[:, 1],  
 marker='v', label='Actual', s=200)  
plt.scatter(data\_mean[:, 0], data\_mean[:, 1],  
 alpha=0.75, label='Mean', marker='^')  
plt.scatter(data\_median[:, 0], data\_median[:, 1],  
 alpha=0.75, label='Median', marker='s')  
plt.scatter(data\_freq[:, 0], data\_freq[:, 1],  
 alpha=0.75, label='Frequent', marker='d')  
plt.scatter(data\_main[:, 0], data\_main[:, 1], marker='.',  
 label='Const', edgecolors='red', facecolors='none', s=200)  
plt.scatter(data\_KNN[:, 0], data\_KNN[:, 1],  
 marker='p', label='KNN')  
  
plt.legend()  
  
plt.savefig('..\images\IMPUTATION.jpg')



OUTPUT

There are four methods that you can use with a normal imputer.

* Mean: Fills the mean of the column within a numerical dataset
* Median: Fills the median of the column within a numerical dataset
* Most Frequent: Fills the most freqent value within the dataset.
* Constant: Fills a constatnt provided in fill\_value

There is also a newer imputer you may have notied in the code. This is the KNNImputer or k\_nearest neighbors imputer. This imputer uses the k-Nearest Neighbors algorithm to estiamte a value. You can set the number of neighbors using the n\_neighbors parameter.

Their is a experamental imputer known as the IterativeImputer which you can find in the [scikit-learn API](https://scikit-learn.org/stable/modules/generated/sklearn.impute.IterativeImputer.html#sklearn.impute.IterativeImputer).

### Feature Engineering: What are we working with?

Another import part of preparing data for your model requires building features. This is becuase data provided rarly has a clear division of features to sample making it hard to design the features matrix. The art of filling out the features of a dataset is called **feature engineering**.

First we must acknolege a few types of features:

* *Categoical Data*: This data deals with categories of the data such as the if a person is tall or short
* Text or Lange Features: These features are considered with the words within a dataset. This particualar feature is common in social media and is the bane of youtubers.
* Image Feature: These features are specific to images such as position and color.

#### Categorical Data

Sklearn does have a few useful features when conducting *vectorization*. When working with categorical data you can use a **DictVecorizer**. This will allow you to seek the features rather quickly.

# %%  
from sklearn.feature\_extraction import DictVectorizer  
data = [{'power': 100, 'usage': 31, 'type': 'development'},  
 {'power': 5, 'usage': 23, 'type': 'development'},  
 {'power': 35, 'v': 37, 'type': 'gameing'}]  
  
v = DictVectorizer(sparse=False)  
X = v.fit\_transform(data)  
print('----------------')  
print(X)  
print('----------------')  
print(v.inverse\_transform(X))  
print('----------------')  
print(v.transform({'power': 33, 'usage': 21, 'type': 'gameing'}))  
  
# %%

OUTPUT

from sklearn.feature\_extraction import DictVectorizer...  
 ----------------  
 [[100. 1. 0. 31.]  
 [ 5. 1. 0. 23.]  
 [ 35. 0. 1. 37.]]  
 ----------------  
 [{'power': 100.0, 'type=development': 1.0, 'usage': 31.0}, {'power': 5.0, 'type=development': 1.0, 'usage': 23.0}, {'power': 35.0, 'type=gameing': 1.0, 'usage': 37.0}]  
 ----------------  
 [[33. 0. 1. 21.]]

You can retrive a DictVectorizer’s feature names through the function .get\_feature\_names(). If we run that on our example we get [‘power’, ‘type=development’, ‘type=gameing’, ‘usage’]

#### Language Features

Language Features focus on word counts or how many times a specific word occurs within a dataset. ALthough you could do this by hand sklearn provided the effiecent class called **CountVecorizer**. This can take a sequence of items, or text file to tokenize the strings.

As an example we will use the descriptions of these youtube videos [EGR491-SAMPLE\_DATA](https://youtube.com/playlist?list=PLZK8Egerqxghhe0xH6_kLpGWg2x8zTgXi).

# %%  
import pandas as pd  
from sklearn.feature\_extraction.text import CountVectorizer  
f = open(  
 '..\CODE\INPUT\COUNTVEC\_TEST.txt', 'r')  
content = f.read()  
content = str.split(content, '>')  
cv = CountVectorizer()  
fit = cv.fit\_transform(content)  
wordBYword = pd.DataFrame(fit.toarray(), columns=cv.get\_feature\_names())  
# %% VECTORIZE BY 2-5 WORD SEQUENCE  
cv2 = CountVectorizer(ngram\_range=(2, 5), analyzer='word')  
fit = cv2.fit\_transform(content)  
twoToFive = pd.DataFrame(fit.toarray(), columns=cv2.get\_feature\_names())  
print(wordBYword)  
print(twoToFive)

also always as asmr be beats believe communicate computers \  
 0 0 0 0 0 1 0 1 0 2  
 1 0 0 0 0 1 0 0 0 0  
 2 1 1 1 1 0 1 0 1 0  
  
 decoder ... this to transmit using via video watch we will you  
 0 0 ... 1 0 0 0 0 1 0 0 2 1  
 1 0 ... 1 0 0 0 0 0 0 1 1 0  
 2 1 ... 1 3 1 1 1 1 1 0 2 3  
  
 [3 rows x 55 columns]  
 also show also show you also show you how also show you how to \  
 0 0 0 0 0  
 1 0 0 0 0  
 2 1 1 1 1  
  
 always to always to some always to some nice \  
 0 0 0 0  
 1 0 0 0  
 2 1 1 1  
  
 always to some nice musical as always as always to ... \  
 0 0 0 0 ...  
 1 0 0 0 ...  
 2 1 1 1 ...  
  
 you how to communicate you how to communicate using you know \  
 0 0 0 0  
 1 0 0 0  
 2 1 1 1  
  
 you know the you know the thing you know the thing that you watch \  
 0 0 0 0 0  
 1 0 0 0 0  
 2 1 1 1 1  
  
 you watch mrbeast you watch mrbeast or you watch mrbeast or pewdiepie  
 0 0 0 0  
 1 0 0 0  
 2 1 1 1  
  
 [3 rows x 278 columns]

You may have notice that this model tends to weigh popular words in these video descriptinos such as “also” or “you”. This is where the **Tfidfvectorizer** comes in handy. Lets look at a similar example with TfidVectorizer instead.

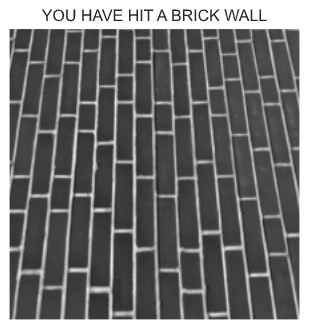
# %%  
  
import pandas as pd  
from sklearn.feature\_extraction.text import TfidfVectorizer  
f = open(  
 '..\CODE\INPUT\COUNTVEC\_TEST.txt', 'r')  
content = f.read()  
content = str.split(content, '>')  
cv = TfidfVectorizer()  
fit = cv.fit\_transform(content)  
wordBYword = pd.DataFrame(fit.toarray(), columns=cv.get\_feature\_names())  
print(wordBYword)  
  
# %%

also always as asmr be beats believe \  
 0 0.000000 0.000000 0.000000 0.000000 0.211054 0.000000 0.277511  
 1 0.000000 0.000000 0.000000 0.000000 0.179873 0.000000 0.000000  
 2 0.140933 0.140933 0.140933 0.140933 0.000000 0.140933 0.000000  
  
 communicate computers decoder ... this to transmit \  
 0 0.000000 0.555021 0.000000 ... 0.163902 0.000000 0.000000  
 1 0.000000 0.000000 0.000000 ... 0.139688 0.000000 0.000000  
 2 0.140933 0.000000 0.140933 ... 0.083237 0.422798 0.140933  
  
 using via video watch we will you  
 0 0.000000 0.000000 0.211054 0.000000 0.000000 0.327805 0.211054  
 1 0.000000 0.000000 0.000000 0.000000 0.236512 0.139688 0.000000  
 2 0.140933 0.140933 0.107183 0.140933 0.000000 0.166474 0.321549  
  
 [3 rows x 55 columns]

#### Image Featuers

Image proccessing is rather difficult as it has a low of data. An example is if you had a 256px by 256px image. Each of the pixels has a color value (usally 4 digits representing red, green, blue and alpha), and positon value. This would be very tedious luckally there is Scikit-image which can analize this images for us. Below is a short example of how to use this module.

# %% SKIMAGE\_TEST  
import os  
import matplotlib.pyplot as plt  
import numpy as np  
import seaborn as sns  
import skimage.data  
from skimage import color, data, feature  
  
sns.set()  
  
  
image = color.rgb2gray(data.brick())  
plt.axis('off')  
plt.imshow(image, cmap='gray')  
plt.title('YOU HAVE HIT A BRICK WALL')



BRICK WALL

You may notice that Scikit-image works as both a image analyzer and manipulator. This feature through will be expalin further in the [Application](#X4f41e06d1c7ba6ac291f933ab9e02168210fdc5) section of this project.

#### Derived Features

Derieved Features are purly numerical features. Scikit-learn treat all numical feature inputs as derived features by deafult, but there is more than you can do. These can be infered with common models like **LinearRegression** and **PolynomialFeatures**.

### Picking a Model: What will you use?

Now that we have clean data we are able to pick a class of model we wish to use. From the (Scikit-learn API)[https://scikit-learn.org/stable/modules/classes.html] you can find all classes of models. You class should include what type of information you wish to get, and what information you have.

After you have chose your model class there are two things you must do next. You must select your **hyperparamters** or the “knobs” which your ML algorthm can adjust, and you must **validate** your model.

##### Hyperparemters

Hyperparemtesr are what makes a ML algorthim tick. These are specific to a model. Forinstance the *KNeighborsClassifier* has a parameter *n\_neighbors* while the *PolynomicalFeatures* model has a *degree* features.

These parameters will determin how your algorithm will solve the problems and will be fill out when instantiating the model.

##### Validation

Once you have selected your hyperparametes you must make a decision if you could improve the estimations. This is where model validation begins to work.

The first rule of model validation is that your model **cannot** be trained on the same data that it is validated on. This would give your model a constant 100% acurracy which most likly is not true. There are a few methods to evaluate your model but some of the most common are *holdout sets* and *cross validation*

Holdout sets work by spliting the data, training on one part of the data, and validating on the other set. This is all facilited by the class train\_test\_split and an example of how this works cna be found here

# IMPORTS  
import matplotlib.pyplot as plt # PLOTING  
import numpy as np # NUMPY  
import pandas as pd # PANDAS  
from sklearn.datasets import load\_iris # Dataset  
from sklearn.metrics import accuracy\_score # Metrics for analysis  
# Model Validation Tools  
from sklearn.model\_selection import train\_test\_split  
  
# Models  
from sklearn.neighbors import KNeighborsClassifier  
from sklearn.neighbors import RadiusNeighborsClassifier  
  
iris = load\_iris()  
X = iris.data  
y = iris.target  
  
## MODEL SETUP  
model\_1 = KNeighborsClassifier(n\_neighbors=2)  
model\_1.fit(X, y)  
y\_model = model\_1.predict(X)  
  
print(accuracy\_score(y, y\_model)) # <-- 0.98

This particular model got an accuracy score of 0.98 (This is similar to the

of a trendline within Excel).

But holdoutsets are not the most efficent as some of the data is only used for one part of setting up the model. Thus cross-validation tends to be the method of choice. Cross validation fits a model to each section of data and validates it against all the others. Below is an example of cross validation

#### CROSS VALIDATION ####  
from sklearn.model\_selection import cross\_val\_score  
score = cross\_val\_score(model\_1, X, y, cv=5)  
score.mean()

This model got a score of 0.96 based on the data. You may notice the *cv* parmeter. This sets the number of splits the data will go through. If you were to output score it would look like

array([0.96666667, 0.93333333, 0.93333333, 0.9 , 1. ])

You then take the average to see the general accurace of the model.

##### Picking The Best Model

In some cases you may notice your model is not performing properly.

* You could pick a differrent model.
* You could gather more datapoints.

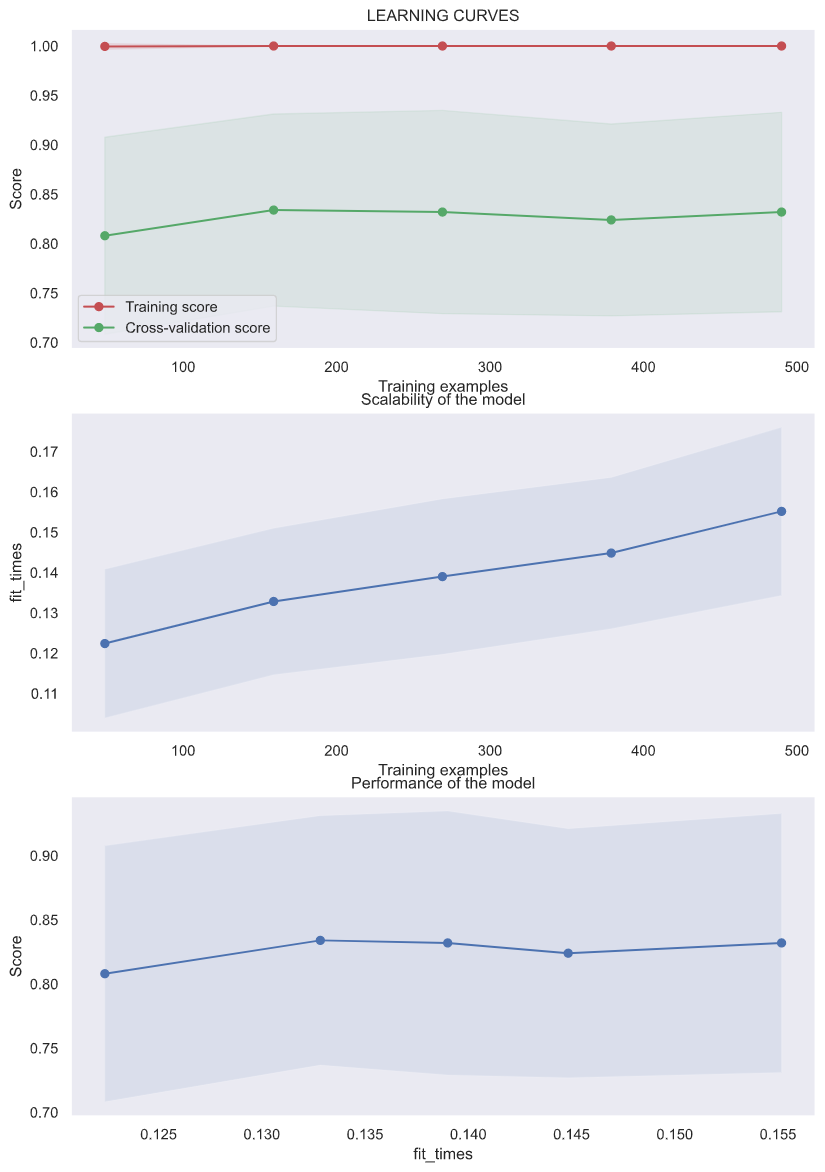
These each have a traidoff and you must consider the complexity of your current model and the amount of data your have. Sadly no model will be perfic so you must find the best point between *bias* and *varience*.

If you model has high *bias* it tends to **underfit** your data meaning its model does not adiqatly shape to your data. The oposite result of **Overfitting** where your model is now accounting for errors and randomness within the data, tends to be a resul of high *varience.*

This is where the **coefficen of determination** or R^2 value comes in handy. This number usually randes from 1 to 0, a R^2 of one means the model matches the data while one of 0 means your model matches at most the mean of your datapoints. BUt there is an interesting result of validation that helps us notice this, the bias and varience changes with the similatity in perfomance between the training and validation sets. This gives us ways to programicly look at the data to see what we can do.

The best way to evaluate the bias and vaience of a model cna be acomplised by a *validation curve*. Scikit-learn includes a object for this called *learning curve*. Below is an example of how to use this method.

from sklearn.model\_selection import learning\_curve  
from sklearn.datasets import make\_blobs  
from sklearn.ensemble import RandomForestClassifier as RandForClassy  
import numpy as np  
import mratplotlib.pyplot as plt  
import seaborn as sns  
sns.set()  
  
### HELPER FUNCTION FROM SCIKIT LEARN  
def plot\_learning\_curve(Esitmator, X, y, axes=None, cv=5, n\_jobs=None, train\_sizes=np.linspace(.1, 1.0, 5)):  
 if axes is None:  
 \_, axes = plt.subplots(1, 3, figsize=(20, 5))  
  
  
 axes[0].set\_xlabel("Training examples")  
 axes[0].set\_ylabel("Score")  
  
 train\_sizes, train\_scores, test\_scores, fit\_times, \_ = \  
 learning\_curve(estimator, X, y, cv=cv, n\_jobs=n\_jobs,  
 train\_sizes=train\_sizes,  
 return\_times=True)  
 train\_scores\_mean = np.mean(train\_scores, axis=1)  
 train\_scores\_std = np.std(train\_scores, axis=1)  
 test\_scores\_mean = np.mean(test\_scores, axis=1)  
 test\_scores\_std = np.std(test\_scores, axis=1)  
 fit\_times\_mean = np.mean(fit\_times, axis=1)  
 fit\_times\_std = np.std(fit\_times, axis=1)  
  
 # Plot learning curve  
 axes[0].grid()  
 axes[0].fill\_between(train\_sizes, train\_scores\_mean - train\_scores\_std,  
 train\_scores\_mean + train\_scores\_std, alpha=0.1,  
 color="r")  
 axes[0].fill\_between(train\_sizes, test\_scores\_mean - test\_scores\_std,  
 test\_scores\_mean + test\_scores\_std, alpha=0.1,  
 color="g")  
 axes[0].plot(train\_sizes, train\_scores\_mean, 'o-', color="r",  
 label="Training score")  
 axes[0].plot(train\_sizes, test\_scores\_mean, 'o-', color="g",  
 label="Cross-validation score")  
 axes[0].legend(loc="best")  
  
 # Plot n\_samples vs fit\_times  
 axes[1].grid()  
 axes[1].plot(train\_sizes, fit\_times\_mean, 'o-')  
 axes[1].fill\_between(train\_sizes, fit\_times\_mean - fit\_times\_std,  
 fit\_times\_mean + fit\_times\_std, alpha=0.1)  
 axes[1].set\_xlabel("Training examples")  
 axes[1].set\_ylabel("fit\_times")  
 axes[1].set\_title("Scalability of the model")  
  
 # Plot fit\_time vs score  
 axes[2].grid()  
 axes[2].plot(fit\_times\_mean, test\_scores\_mean, 'o-')  
 axes[2].fill\_between(fit\_times\_mean, test\_scores\_mean - test\_scores\_std,  
 test\_scores\_mean + test\_scores\_std, alpha=0.1)  
 axes[2].set\_xlabel("fit\_times")  
 axes[2].set\_ylabel("Score")  
 axes[2].set\_title("Performance of the model")  
  
 return plt  
  
  
fig, axes = plt.subplots(3, 1, figsize=(10, 15))  
  
  
# Cross validation with 100 iterations to get smoother mean test and train  
# score curves, each time with 20% data randomly selected as a validation set.  
axes[0].set\_title('LEARNING CURVES')  
estimator = RandForClassy()  
plot\_learning\_curve(estimator, X, y, axes=axes,  
 cv=50, n\_jobs=4)  
  
  
plt.show()  
# %%

And the output comes out to be 

In general the optimal model will be where the distance between the validatoin and training score is near 0. In our example that would be around 300 estimators. But this moel is still rather underfit for this cisumstance.

Thus we are left with the decision to change our model or get more data. As this was generate for a random set of data for the section [Special Topic: Random Forest](#X2052198495ce7db9bce7fa4ed72384b2c25caf4) we will not dive much deeper into this dataset.

### Piplines

As you can see above there is a lost that goes into setting up a model. To make your code leaner you can use a **pipline** this streamlined method of managing a model allows you to use an impuerter, and model which will be stored as a pipline object and will automatically run through the whole proccess you have setup upon call to the .fit() fucntion.

from sklearn.pipeline import make\_pipeline  
from sklearn.impute import SimpleImputer  
from sklearn.preprocessing import PolynomialFeatures  
from sklearn.linear\_model import LinearRegression  
import numpy as np  
from numpy import nan  
model = make\_pipeline(SimpleImputer(strategy='mean'),  
 PolynomialFeatures(degree=2),  
 LinearRegression())  
X = np.array([[nan, 0, 3],  
 [3, 7, 9],  
 [nan, 5, nan],  
 [4, nan, 6],  
 [nan, 8, 1],  
 [1, 1, 1],  
 [3, 4, 9]])  
y = np.array([1,2,4,8,16,32,64])  
model.fit(X, y)  
print(y) #[ 1 2 4 8 16 32 64]  
print(model.predict(X))#[ 1. 2. 4. 8. 16. 32. 64.]

### 3.4. Special Topic: Random Forests

There is a unique form of model called the **random forests** model. This model is is an *ensemble learning* which uses simple dicision trees to calcualte its outcome.

A decision tree is a type of binary tree that uses a seires of decisions to select it out. This could be used to output your data in a unique manner. Below is an example of a simple decision tree to find the quadrant of the Cartesian plane a specific data point is in:

graph TB  
 A[Is x > 0] -->|YES| B(Is y > 0)  
 A -->|NO| C(Is y > 0)  
 B -->|YES| D(ONE)  
 B -->|NO| E(FOUR)  
 C -->|YES| F(TWO)  
 C -->|NO| G(THREE)

Firure 1: Decision tree example

Esentially a Random Forest creates this decision tree for us. This makes random forests a excelent unserpervised classification model.

You can build the dicision tree through a *DecisionTreeClassifier* which is under the sklearn.tree section. But when fitting you must be careful to avoid Over-fitting. Over-fitting is when your model becomes “Attached” to your data and will look at the specific data points. Luckally there is a simple solution. Use multiple instances of your model fit to similar or the same data points and use all of them to make the final decision. The second method with Decision Tree models is to uses the ensemle method *bagging* which quickly creates multiple parrael decsion trees of data. The group of random decision trees is where we get random forests. Luckally sklearn.ensemble include the RnadomForestClassifier estimator which easily helps your form a random forest rapidly.

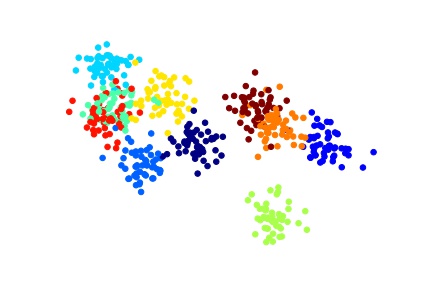
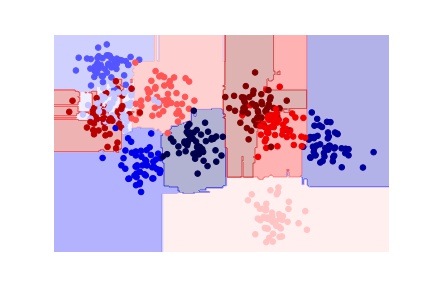
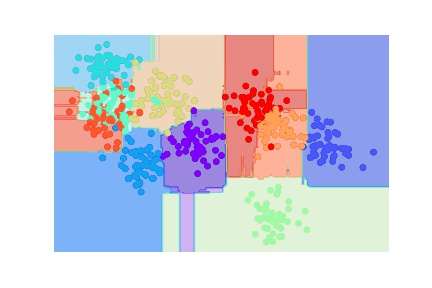
In addtion to classification using a RandomForestClassifier you can use a RnadomForestRegressor for regression. This type of model is excelent at detecting oscilating functions and does not need much instructions on that front.

Below is an example of how to use a RandomForestClassifier:

CODE:

# %% Setup  
from sklearn.datasets import make\_blobs  
from sklearn.ensemble import RandomForestClassifier as RandForClassy  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
sns.set()  
  
# %% Getting Data  
  
X, y = make\_blobs(500, 2, centers=10, cluster\_std=1, random\_state=1892)  
ax = plt.ax  
plt.scatter(X[:, 0], X[:, 1], c=y, cmap='jet')  
plt.savefig('..\images\RAND\_FOREST-CLASS-DATA.jpg')  
# %% Helper Function From Text  
  
  
def visualize\_classifier(model, X, y, ax=None, cmap='rainbow'):  
 ax = ax or plt.gca() # Set the Plot axis  
  
 # Plot the training points  
 ax.scatter(X[:, 0], X[:, 1], c=y, s=30, cmap=cmap,  
 clim=(y.min(), y.max()), zorder=3) # Create a scatter plot of the data  
 ax.axis('tight') # Set the axis Range to tight  
 ax.axis('off') # Turn Off Axis Desplay  
 xlim = ax.get\_xlim() # Get The X/Y LIMITS  
 ylim = ax.get\_ylim()  
  
 # fit the estimator  
 model.fit(X, y) # Fit the model  
 xx, yy = np.meshgrid(np.linspace(\*xlim, num=200),  
 np.linspace(\*ylim, num=200)) # Make grid of datapoints  
 Z = model.predict(np.c\_[xx.ravel(), yy.ravel()]).reshape(  
 xx.shape) # Use model to predict data  
  
 # Create a color plot with the results  
 n\_classes = len(np.unique(y))  
 contours = ax.contourf(xx, yy, Z, alpha=0.3,  
 levels=np.arange(n\_classes + 1) - 0.5,  
 cmap=cmap, clim=(y.min(), y.max()),  
 zorder=1)  
  
 ax.set(xlim=xlim, ylim=ylim)  
  
  
# %% Setting UP and Running Module  
  
model = RandForClassy(n\_estimators=200)  
visualize\_classifier(model, X, y, cmap='seismic')  
plt.savefig('..\images\RAND\_FOREST-CLASS-MODEL\_200.jpg')  
model = RandForClassy(n\_estimators=500)  
visualize\_classifier(model, X, y, cmap='rainbow')  
plt.savefig('..\images\RNAD\_FOREST-CLASS-MODEL\_500.jpg')  
# %%

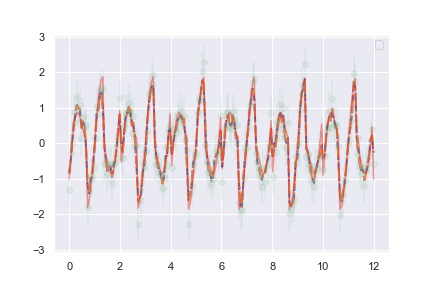
The output looks like this

If you zoom into the image files you might notice that these models are albe to show a reagon to show the teal category that mixes within the red, yellow and blue categories. Sadly due to the image quality it is hard to tell how well the 500 group and 200 group system worked.

In the Electronic Warfare (EW) industy analyzing signals through noise is neccesary. Therefore a unquie cabability for Cognitive EW (CEW) is achived via Rand Forest Regression.

### CEW EXAMPLE ###  
# %% Setup  
import matplotlib.pyplot as plt  
import numpy as np  
import seaborn as sns  
from scipy.signal import sawtooth  
from sklearn.datasets import make\_blobs  
from sklearn.ensemble import RandomForestRegressor as RandForRegy  
  
sns.set()  
rng = np.random.RandomState(420)  
x = np.linspace(0, 12, 200)  
# SIGNAL FUNCTION  
  
  
def signal(x, noise\_mult=1):  
 base = np.sin(2\*np.pi\*x)  
 saw = sawtooth(3\*np.pi\*x)  
 noise\_a = 0.5 \* (np.random.randint(-90, 90, len(x))/100)  
 noise\_b = 0.2 \* np.random.randint(0, 1) \* np.cos(3\*sawtooth(x)+base)  
 noise = (noise\_a + noise\_b)  
 return base + saw + noise\*noise\_mult  
  
  
y = signal(x)  
  
  
# %% SETTING UP/RUNNING OUT MODEL  
model\_200 = RandForRegy(200)  
model\_500 = RandForRegy(500)  
  
model\_200.fit(x[:, None], y)  
model\_500.fit(x[:, None], y)  
  
y\_fit\_200 = model\_200.predict(x[:, None])  
y\_fit\_500 = model\_500.predict(x[:, None])  
# %% FILL OUT  
  
  
plt.plot(x, y\_fit\_200, '--', linewidth=2, label='200\_fit')  
plt.plot(x, y\_fit\_500, '-.', linewidth=2, label='500\_fit')  
plt.plot(x, signal(x, noise\_mult=0), alpha=0.4, c='red', label='True')  
plt.errorbar(x, y, 0.5, fmt='o', alpha=0.2, label='Noisy')  
plt.legend(loc='best')  
plt.savefig('..\images\RAND\_FOREST-REG-ALL.png')  
# %%



Data Output

You may notice that despite the noise the data actually does get the underlying siganl despite the noise.

## 4. Project: Face Detection Pipeline

There are a few classical examples of Machine Learning. One is making an algorithem that can recognize hadwriten digits and anotheris to write a faical recoginition program in this program we will test this out.

### 4.1. Output

### 4.2. Code

# %% IMPORTS  
from skimage.io import imread  
from itertools import chain  
  
import matplotlib.pyplot as plt  
import numpy as np  
import skimage.data  
from skimage import color, data, feature, transform  
import sklearn  
from sklearn.datasets import fetch\_lfw\_people  
from sklearn.ensemble import RandomForestClassifier as RandForClassy  
from sklearn.feature\_extraction.image import PatchExtractor  
from sklearn.model\_selection import GridSearchCV, cross\_val\_score  
from sklearn.naive\_bayes import GaussianNB  
from sklearn.svm import LinearSVC  
  
# %% GET FALSE DATA  
imgs\_to\_use = ['camera', 'text', 'coins', 'moon',  
 'page', 'clock', 'immunohistochemistry',  
 'chelsea', 'coffee', 'hubble\_deep\_field']  
images = [color.rgb2gray(getattr(data, name)())  
 for name in imgs\_to\_use]  
# %% GET TRUE  
faces = fetch\_lfw\_people()  
positive\_patches = faces.images  
# %% FUNCTION DEFINITIONS  
  
  
def extract\_patches(img, N, scale=1.0, patch\_size=positive\_patches[0].shape):  
 extracted\_patch\_size = tuple((scale \* np.array(patch\_size)).astype(int))  
 extractor = PatchExtractor(patch\_size=extracted\_patch\_size,  
 max\_patches=N, random\_state=0)  
 patches = extractor.transform(img[np.newaxis])  
 if scale != 1:  
 patches = np.array([transform.resize(patch, patch\_size)  
 for patch in patches])  
 return patches  
  
  
def sliding\_window(img, patch\_size=positive\_patches[0].shape,  
 istep=2, jstep=2, scale=1.0):  
 Ni, Nj = (int(scale \* s) for s in patch\_size)  
 for i in range(0, img.shape[0] - Ni, istep):  
 for j in range(0, img.shape[1] - Ni, jstep):  
 patch = img[i:i + Ni, j:j + Nj]  
 if scale != 1:  
 patch = transform.resize(patch, patch\_size)  
 yield (i, j), patch  
  
  
negative\_patches = np.vstack([extract\_patches(im, 1000, scale)  
 for im in images for scale in [0.5, 1.0, 2.0]])  
  
# %% EXTRACT HOG FEATURES  
X\_train = np.array([feature.hog(im)  
 for im in chain(positive\_patches,  
 negative\_patches)])  
y\_train = np.zeros(X\_train.shape[0])  
y\_train[:positive\_patches.shape[0]] = 1  
  
# %%  
grid = GridSearchCV(LinearSVC(max\_iter=100000), {'C': [1.0, 2.0, 4.0, 8.0]})  
grid.fit(X\_train, y\_train)  
  
model = grid.best\_estimator\_  
model.fit(X\_train, y\_train)  
# %% RUN FIMPLE TEST IMAGE  
  
  
def prepImg(img):  
  
 test\_image = img  
 test\_image = skimage.color.rgb2gray(test\_image)  
 test\_image = skimage.transform.rescale(test\_image, 0.5)  
 test\_image = test\_image[:160, 40:180]  
  
 return test\_image  
# %% PLOT TEST IMAGE  
  
  
def pltImg(img):  
 img = prepImg(img)  
 indices, patches = zip(\*sliding\_window(img))  
 patches\_hog = np.array([feature.hog(patch) for patch in patches])  
 patches\_hog.shape  
  
 labels = model.predict(patches\_hog)  
 labels.sum()  
  
 fig, ax = plt.subplots()  
 ax.imshow(img, cmap='gray')  
 ax.axis('off')  
  
 Ni, Nj = positive\_patches[0].shape  
 indices = np.array(indices)  
 valid\_indicies = indices[labels == 1]  
 for i, j in valid\_indicies:  
 ax.add\_patch(plt.Rectangle((j, i), Nj, Ni, edgecolor='red',  
 alpha=0.2, lw=2, facecolor='none'))  
 mean\_j = np.mean(valid\_indicies, axis=1)  
 mean\_i = np.mean(valid\_indicies, axis=0)  
  
 print(mean\_i)  
  
 return plt  
  
  
pltImg(skimage.data.astronaut())  
  
# %%  
exit = False  
  
while(not exit):  
 file = input('Enter a File Path:')  
 print('Prepring Display')  
 pltImg(imread(file))  
 plt.show()  
 if(input('Woud you like to coninue [Y/n]') == 'n'):  
  
 print('Goodbye')  
 exit = True  
  
# %%