# 1. Supervised Learning with scikit-learn

May-01-2022

- ML = ⊙ Giving computers the ability to learn to make decisions from data
  ⊙ without being explicitly programmed
- Supervised learning = uses labeled data
- Unsupervised learning = ⊙ uses unlabeled data
  ⊙ uncovering hidden patterns from unlabeled data
  eg grouping customers in to distinct categories based on purchasing behavior
  (clustering)
- Reinforcement learning = ⊙ interacts with an environment
  ⊙ learn how to optimize their behavior
  ⊙ Given a system of rewards and punishment
  ⊙ Draws inspiration from behavioral psychology

## Supervised Learning

⊙ predictor variables ($x$) = Features = independent variable

⊙ Target variable ($y$) = dependent variable = response variable
  ↳ Classification: categorical variable
  ↳ Regression: continuous variable or float

## 2 Exploratory Data Analysis

Iris dataset

```
from sklearn import datasets
import pandas as pd
  "     numpy  "  np
  "     matplotlib.pyplot as plt
plt.style.use('ggplot')
iris = datasets.load_iris()
type(iris) ⟶ Bunch ⟶ similar to dictionary
type(iris.data), type(iris.target)
```

```
iris.~~data~~.shape
iris.target_names
```

EDA

```
X = iris.data
y = iris.target
df = pd.DataFrame(X, columns=iris.feature_names)
print(df.head())
```

```
knn = KNeighborsClassifier(n_neighbors = 8)
knn.fit (X_train, y_train)
y_pred = knn.predict (X_test)
print (\" Test set predictions: \\n { }\".format(y_pred))
```

# Accuracy

```
knn.score(X_test, y_test)
```

---

## Introduction to regression

- Target is continuous variable

```
boston = pd.read_csv('boston.csv')
boston.head()
```

Creating feature and target arrays

```
X = boston.drop ('MEDV', axis=1).values
y = boston ['MEDV'].values
```

Predicting house prices from a single feature

```
X_rooms = X[:, 5]          # no. of room column
type(X_rooms), type(y)
```

Reshaping

```
y = y.reshape (-1, 1)                    #keep first dimension ~~and~~ but
                                         #add another dimension of
X_rooms = X_rooms.reshape(-1, 1)         #size 1
```

Plotting house values v/s no. of rooms

```
plt.scatter(X_rooms, y)
plt.ylabel( 'Value of house /1000 ($)')
plt.xlabel (' Number of rooms')
plt.show()
```

## Linear Regression on all features

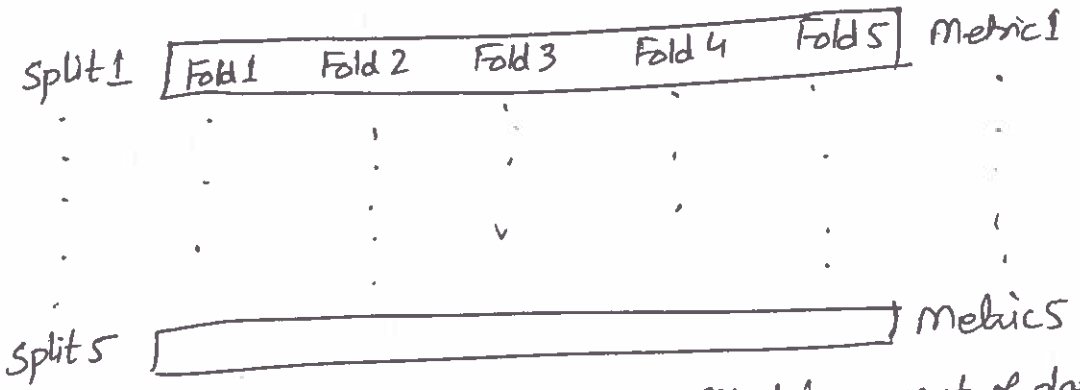
```
from sklearn.model_selection import train_test_split
  "        "   .linear_model        "      LinearRegression

X_train, X_test, y_train, y_test = train_test_split(X, y, testsize=0.3,
                                                    random_state=42)

reg_all = LinearRegression()
reg_all.fit(X_train, y_train)
y_pred = reg_all.predict(X_test)
reg_all.score(X_test, y_test)
```

## Cross Validation

– 5 folds = 5-Fold CV
K folds = k-Fold CV

**Motivation**

- Model performance is dependent on the way data is split
- Not representative of the model's ability to generalize



- Divide data in to 5 Fold ⟶ Step 1 → fit data on rest of data and use Fold 1 as test
  → repeat for every Fold.

```
from sklearn.model_selection import cross_val_score
  "        "   linear_mode         "      LinearRegression

reg = LinearRegression()
cv_results = cross_val_score(reg, X, y, cv=5)
print(cv_results)
np.mean(cv_results)
```

## Lasso for feature selection in scikit-learn

```
from sklearn.linear_model import Lasso
names = boston.drop('MEDV', axis=1).columns.
lasso = Lasso(alpha=0.1)
lasso_coeff = lasso.fit(X, y).coeff_
_ = plt.plot(range(len(names)), lasso_coeff)
_ = plt.xticks(range(len(names)), names, rotation=60)
_ = plt.ylabel('Coefficients')
plt.show()
```

## Fine tunin your Model

### How good is your model

### Classification metrics

- Accuracy is not always a useful metric

### Case imbalance examples: Emails

- Spam classification
  - 99% of emails are real
  - 1% of emails are spam
- Buildin a model that predicts All emails are real will be 99% accurate. but horrible at actually classifying spam

class of interest is +ve class

### Diagnosing classification predictions

① Confusion Matrix

|  | Predicted spam | ~~Actu~~ Predicted real |
|---|---|---|
| Actual spam | TP | FN |
| Actual real | FP | TN |

$$\text{Accuracy} = \frac{tp+tn}{tp+tn+fp+fn}$$

$$\text{Precision} = \frac{tp}{tp+fp} = PPV$$

PPV → Positive predicti value

$$\text{Recall} = \frac{tp}{tp+fn}$$

→ sensitivity
Hit rate
TPR → true positive rate

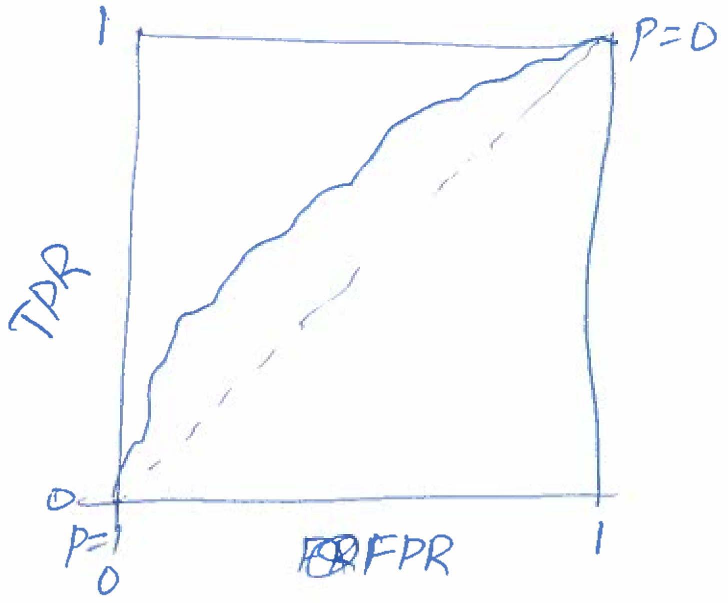
① F1score: $2 \cdot \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$

↓
harmonic mean of precision and recall

## Probability threshold

- By default, logistic Regression threshold = 0.5
- ~~All classifiers~~ Not specific to logreg
  - ⊙ k-NN also has a threshold
  - ⊙ What happens if we vary this threshold

## The ROC-Curve (Receiver Operating Characteristic Curve)



$= \frac{TP}{P} = 1 - FNR$

TPR = sensitivity = recall

↳ probability of detection

FPR = probability of false alarm

↳ fall-out

$FPR = \frac{FP}{N} = \frac{FP}{FP+TN}$ ~~= (1-TPR)~~

## Plotting ROC-curve

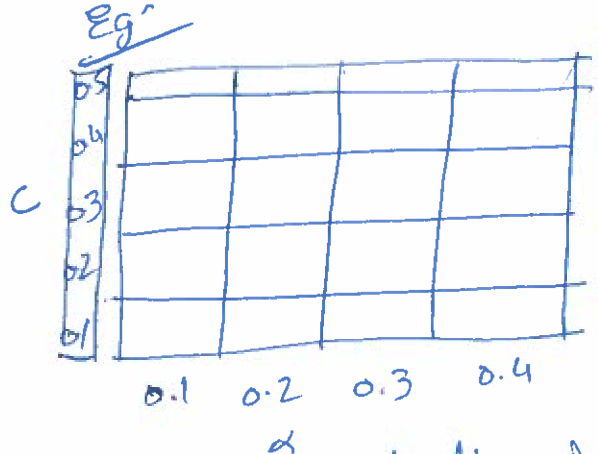
```
from sklearn.metrics import roc_curve
y_pred_prob = logreg.predict_proba(X_test)[:,1]
fpr, tpr, thresholds = roc_curve(y_test, y_pred_prob)
plt.plot([0,1], [0,1], 'k--')
plt.plot(fpr, tpr, label='Logistic Regression')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive   "   ')
plt.title('Logistic Regression ROC Curve')
plt.show()
```

## Choosing Correct Hyperparameter

- Try bunch of different hyperparameters values
- Fit all of them separately
- See how well each performs
- Choose the best performing one
- It is essential to use cross-validation

## Grid Search cross-validation

Eg:



pick best set of alpha an C

```
from sklearn.model_selection import GridSearchCV
param_grid = {'n_neighbors': np.arange(1,50)}
knn = KNeighborsClassifier()
knn_cv = GridSearchCV(knn, paramgrid, cv=5)
knn_cv.fit(X,y)
knn_cv.best_param_
knn_cv.best_score_
```

## Hold-out set for final evaluation

- How well can the model perform on never before seen data?
- using ALL data for cross-validation is not ideal.
- Split data into training and hold-out set at the begining.
- Perform grid search cross-validation on training set.

```
df_origin = df_origin.drop('origin_Asia', axis=1)
print(df_origin.head())
```

Linear Regression ~~using~~ with dummy variables

```
from sklearn.model_selection import train_test_split
 "        "      .linear_model import Ridge.
X_train, X_test, y_train, y_test = train_test_split(X, y,
                                        test_size=0.3,
                                        random_state=42)
ridge = Ridge(alpha=0.5, normalize=True).fit(X_train,
                                        y_train)
ridge.score(X_test, y_test)
```

---

## Handling Missing Data

```
df = pd.read_csv('diabetes.csv')
df.info()
```

#Dropping missing data

```
df.insulin.replace(0, np.nan, inplace=True)
df.triceps.  "   (0, np.nan,    "     )
df.bmi. replace ( "     "       "     )
df.info()
```

# One way

```
df.dropna()     #→ dropping all rows with missing data.
df.shape        # not good.
```

# another option

```
# make an educated guess...
# eg:- Using mean of the non-missing entries.
from sklearn.preprocessing import Imputer.
imp = Imputer(missing_values='NaN', strategy='mean', axis=0)
imp.fit()
X = imp.transform(X)
```

## Ways to normalize your data

- Standardization: subtract the mean & divide by variance
- All features are centered around zero and have variance one
- Can also subtract the minimum and divide by the range
- Minimize zero and maximum one.
- Can also normalize so the data ranges from -1 to +1
- ~~see scikit-learn docs for fu~~

## Scaling in sklearn

```
from sklearn.preprocessing import scale
X_scaled = scale (X)
np.mean (X) , np.std (X)
np.mean (X_scaled) , np.std (X_scales)
```

## Scaling in a pipeline

```
from sklearn.preprocessing import StandardScaler
Steps = [('scaler', StandardScaler ()),
         ('knn' , kNeighborsClassifier ())]
pipeline = Pipeline( steps)
~~pipeline.~~
X_train, X_test, y_train, y_test = train_test_split (X, y, test_size=0.2,
                                            random_state=21)
knn_scaled = pipeline.fit(X_train , y_train)
y_pred = pipeline.predict(X_test)
accuracy_score(y_test, y_pred)
knn_unscaled = KNeighborsClassifier ().fit (X_train, y_train)
knn_unscaled.score (X_test, y_test)
```

## 2-1 ② Unsupervised Learning

Clustering for dataset Exploration

### k-means Clustering

- Finds clusters of samples
- Number of clusters must be specified
- Implemented in sklearn ("scikit-learn")

```
from sklearn.cluster import KMeans

model = KMeans(n_clusters=3)
model.fit(samples)
labels = model.predict(samples)
labels.
```

### Cluster labels for new samples

- New samples can be assigned to existing clusters.
- k-means remembers the mean of each cluster (the centroids)
- Finds the nearest centroid to each new sample.

```
new_samples
new_labels = model.predict(new_samples)
```

### Scatter plots

```
import matplotlib.pyplot as plt
xs = samples[:,0]
ys = samples[:,2]
plt.scatter(xs, ys, c=labels)
plt.show()
```

### Evaluating Cluster

- Measure quality of clustering.
- Informs choice of how many clusters to look for

## Transforming features for better clustering

### Piedmont wine dataset

### Clustering the wines

```
from sklearn.cluster import KMeans
model = KMeans(n_clusters = 3)
labels = model.~~fit (sam~~fit_predict(samples)
```

### Clusters v/s varieties

```
df = pd.DataFrame({'labels': labels,
                   'varieties': varieties})
ct = pd.crosstab(df['labels'], df['varieties'])
ct
```

### Feature variances

- The wine features have very different variances.
- Variance of a feature measure spread of its values.

### Standard Scalar

- In kmeans: feature variance = feature influence
- StandardScalar transforms each feature to have mean 0 and variance 1
- Features are said to be "standardized"

```
from sklearn.preprocessing import StandardScalar
scaler = StandardScaler()
Scalar.fit(samples)
StandardScalar(copy=True, with_mean=True, with_std=True)
samples_scaled = scaler.transform(samples)
```

### Similar Methods

- StandardScalar and KMeans have similar methods
- Use fit() / transform() with StandardScalar
- Use fit() / predict() " KMeans

# 2-2 Visulizing Hieraichies

- t-SNE: creates a 2D map of a dataset → based on proximity.
- Hierarchial clustering

A hierarchy of groups

- Eurovision dataset
- ~~Counties~~ tree-like diagram = dendrogram
- Group countries into larger and larger clusters.

Steps

- Every country is in a seperate cluster.
- At each step two closer clusters are merged.
- Continue until all countries in a single cluster.
- This is "agglomerative" hierarchial clustering.
- There is also "divisive clustering" which works other way around.

Dendrogram

- Read from bottom up
- vertical lines represent clusters.

Hierarchical Clustering with SciPy

- Given samples (the array of scores), and country_names

```
import matplotlib.pyplot as plt
from scipy.cluster.hierarchy import linkage, dendrogram
mergings = linkage(samples, method = 'complete')
dendrogram(mergings, labels = country_names,
           leaf_rotation = 90, leaf_font_size=6)
plt.show()
```

## Aligning cluster labels with country_names

```
import pandas as pd
pairs = pd.DataFrame({'labels': labels, 'countries': country_name})
print(pairs.sort_values('labels'))
```

## t-SNE for 2-dimensional maps

- t-SNE = "t-distributed stochastic neighbor embedding"
- Maps samples to 2D space (or 3D)

### t-SNE on Iris dataset

- Iris data set is 4D
- t-SNE maps samples to 2D space
- t-SNE didn't know that there were different species
- ... yet kept the species mostly separate.

### Interpreting t-SNE scatter plots

- 'versicolor' and 'virginica' harder to distinguish from one another.
- Consistent with k-means inertia plot: could argue for 2 clusters, or for 3.

### t-SNE in sklearn

Samples ⟶ 2D Numpy array

species ⟶ list

```
import matplot.lib as plt
from sklearn.manifold import TSNE
model = TSNE(learning_rate =100)
transformed = model.fit_transform(samples)
xs = transformed[:,0]
ys = transformed[:,1]
plt.scatter(xs, ys, c=species)
plt.show()
```

# 2-3 Visualizing PCA transformation

## Dimension Reduction

- More efficient storage and computation
- Remove less-informative "noise" features
- ... which cause problems for prediction tasks e.g: classification, regres

## PCA

- Fundamental dimension reduction technique
- First step "decorrelation"
- Second step reduces dimension

## PCA aligns data with axes

- Rotates data samples to be aligned with axes
- Shift data samples so they have mean 0

## PCA follows the fit/transform pattern

- like KMeans or StandardScalar
- fit() learns the transformation from given data
- transform() applies learned transformation
- transform() can be applied to new unseen samples.
- Samples → array of two features (total_phenols & od280)

```
from sklearn.decomposition import PCA
model = PCA()
model.fit(samples)
transformed = model.transform(samples)
transformed
```

- Rows of transformed corresponds to samples
- Columns of transformed are "PCA features".
- Row gives PCA feature values of corresponding sample.

## Versicolor dataset

- "versicolor", one of iris species.
- only 3 features: sepal_length, sepal width, and petal width
- Samples are points in 3D space.
- Versicolor has intrinsic dimension 2
  - ⊙ samples lie close to flat 2-D sheet
  - ⊙ so can be approximated using 2 features

## PCA identifies intrinsic dimensions

- Scatter plots work only if samples have 2 or 3 features.
- PCA identifies intrinsic dimensions when samples have any number of features.
- Intrinsic dimension = number of PCA features with signific variance.
- PCA features are ordered by variance (descending)
- Intrinsic dimension is number of PCA features with significant variance.

## Plotting the variance of PCA features

samples = array of versicolor samples

```
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

pca = PCA()

pca.fit(samples)

features = range(pca.n_components_)
plt.bar(features, pca.explained_variance_)
plt.xticks(features)
plt.ylabels('variance'); plt.xlabel('PCA feature')
plt.show()
```

- pca has reduced the dimension to 2
- Retained the 2 PCA features with highest variance
- Important info. preserved : species remain distinct.
- Discards low variance PCA features.
  - Assumes the high variance features are informative
  - Assumption typically holds in practice (e.g of iris)

## Word frequency arrays

- rows represent documents, columns represent words
- Entries measure presence of each word in each document.
- .... measure using "tf-idf"

## Sparse arrays and csr_matrix

- "Sparse" : most entries are zero.
- can use scipy.sparse.csr_matrix instead on NumPy array
- csr_matrix remembers only the non-zero entries (saves space)

## TruncatedSVD and csr_matrix

* scikit-learn PCA does not support csr_matrix
- use scikit-learn TruncatedSVD instead
- performs same transformation

```
from sklearn.decomposition import TruncatedSVD
model = TruncatedSVD(n_components = 3)
model.fit(documents)     # ~~no. of~~ documents is csr_matrix
transformed = model.transform(documents)
```

## (2-4) Non-Negative Matrix Factorization (NMF)

- like PCA is a dimension reduction technique
- NMF models are interpretable (unlike PCA)
- Easy to interpret means easy to explain

(*) - However, all sample features must be non-negative (>=0)

### Interpretable parts

- NMF expresses documents as combinations of topics (or "themes")
- NMF expresses images as combinations of patterns.

### Using scikit-learn NMF

- Follows fit() / transform() pattern
- Must specify number of components e.g.
  NMF(n_components = 2)
- works with NumPy arrays and with csr_matrix

### Example word-frequency array

- word frequency array has 4 words, many documents
- Measure presence of words in each document using "tf-idf"
  - tf = frequency of word in document
  - idf = reduces influence of frequent words.

### Example Usage of NMF

samples = is the word-frequency array

```
from sklearn.decomposition import NMF
model = NMF(n_components = 2)
model.fit(samples)
nmf_features = model.transformation(samples)
model.components_
nmf_features
```

~~NMF learns~~

## NMF learns interpretable Parts

ex
- Word frequency article (tf-idf)
- 20,000 scientific articles (rows)
- 800 words (columns)

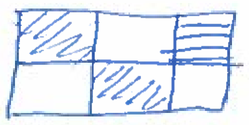
### Applying NMF to the articles

```
articles.shape
from sklearn.decomposition import NMF
nmf = NMF(n_components = 10)
nmf.fit(articles)
nmf.components_.shape
```

- NMF components are topics
- For documents
  - NMF components represents topics
  - NMF features combine topics into documents
- for images, NMF components are parts of images

### Grayscale images

- "Grayscale" image = no colors, only shades of grey
- Measure pixel brightness.
- Represent with value b/w 0 & 1 (0 is black)
- Convert to 2D array.



$\Rightarrow$ [[ 0. 1. 0.5 ]
 [ 1. 0. 1. ]]

- These 2D-arrays can be flattened by enumerating the entries.

## Apply NMF to the word-frequency array

articles is a word frequency array.

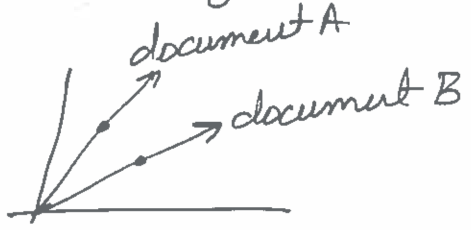
```
from sklearn.decomposition import NMF
nmf = NMF(n_components = 6)
nmf_features = nmf.fit_transform(articles)
```

## Versions of articles

- Different versions of the same document have same topic proportions.
- ... exact feature values may be different
- eg: one version uses many meaningless words.
- But all versions lie on the same line through the origin.

## Cosine similarity

- Uses angle b/w lines
- higher values means more similarity



## Calculating cosine similarity

```
from sklearn.preprocessing import normalize
norm_features = normalize(nmf_features)
# it has index 23
current_article = norm_features [23,:]
similarities = norm_features.dot (current_article)
similarities
```

## DataFrames and labels

- Label similarities with article titles, using a DataFrame
- Titles given as list : titles

```
import pandas as pd
norm_features = normalize(nmf_features)
df = pd.DataFrame (norm_features , index=titles)
current_article = df.loc ['Dog bites man']
similarities = df.dot (current_article)
```

# 3-1 Linear Classifiers

May-6-2022

## Applying logistic Regression and SVM

### Sklearn Refresher

#### Fitting and predicting

```
import sklearn.datasets
newsgroups = sklearn.datasets.fetch_20newsgroups_vectorized()
X, y = newsgroups.data, newsgroups.target
X.shape
y.shape
from sklearn.neighbors import ~~KNearestNe~~ KNeighborsClassifier
knn= KNeighborsClassifier(n_neighbors =1)
knn.fit(X, y)
~~knn~~ y_pred= knn.predict(X)
```

#### Model Evaluation

```
knn.score(X, y)
from sklearn.model_selection import train_test_split
X_train , ~~y~~ X_test, y_train, y_test = train_test_split(X, y)
knn.fit ( X_train , y_train)
knn.score( X_test, y_test)
```

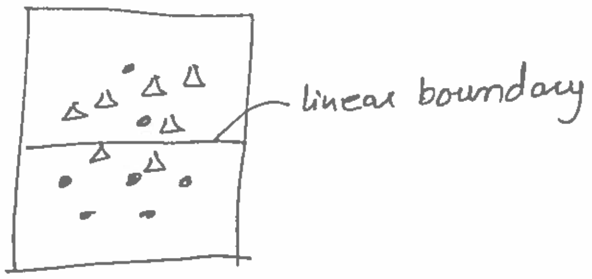
#### Using Logistic Regression

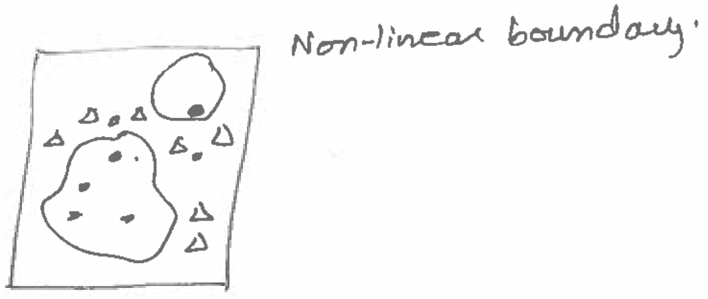
```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(X_train, y_train)
lr.predict (X_test)
lr.score (X_test, y_test)
```

Model Complexity review:

- underfitting : model is too simple, low training accuracy.
- overfitting: model is too complex, low test accuracy.

## Linear decision boundaries





Definitions

- Classification: learning to predict categories
- decision boundary: the surface separating different predicted clas
- linear classifier: a classifier that learns linear decision bounda
  e.g: Logistic Regression, Linear SVC
- linearly seperable: a dataset can be perfectly explained by a linear classifier.

## 3-2 Linear Classifiers : the coefficients

May-6-2022

### Predicting equations

⊙ Dot Product

```
x = np.arange(3)
x          # array([0, 1, 2])
y = np.arange(3,6)
y          # array([3, 4, 5])
x*y           # Element wise multiplication.
array([0, 4, 10])
np.sum(x*y)   == x@y
```

- x@y is called the dot product of x and y, and is written x·y

### Linear Classifier Prediction

- raw model output = coefficients · features + intercept
- compute raw model output, check the sign
  - ⊙ If +ve, predict one class
  - ⊙ If -ve, " other class
- This is same for logistic regression and linear SVM
  - ⊙ fit is different but predict is the same
- Difference in fit relates to loss function (In next chapters)

### How logistic Regression makes predictions

raw model output = coefficients . features + intercepts

```
lr = LogisticRegression()
lr.fit(X, y)
lr.predict(X)[10]
lr.predict(X)[20]
lr.coef_ @ X[10] + lr.intercept_    # raw model output
```

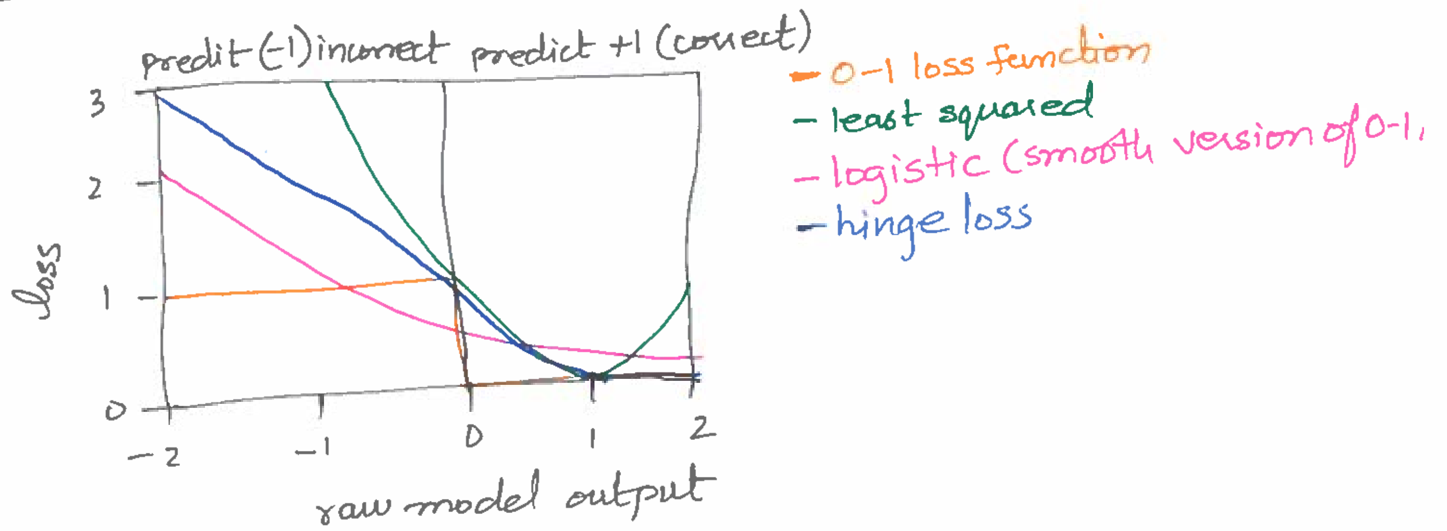
→ array([-33.78 ...]) → It is -ve so we ~~call~~ predict -ve class.

## Minimizing loss

```
from scipy.optimize import minimize
minimize(np.square, 0).x
minimize (np.square, 2).x
```

## Loss functions Diagrams

### The raw model output



May-6-2022 **3-3 Logistic Regression and Regularization**

## How does regularization affect training accuracy

```
lr_weak_reg = LogisticRegression(C=100)
lr_strong_reg = LogisticRegression(C=0.01)
lr_weak_reg.fit(X_train, y-train)
lr_strong_reg.fit(X_train, y_train)
lr_weak_reg.score(X_train,y_train)
lr_strong_reg.score (X_train, y_train)
```

⊛ lower C → smaller coefficients.

⊛ bigger C → bigger coefficients.

regularized loss = original loss + large coefficient penalty

⊙ more regularization: lower training accuracy.

## How does regularization affect test accuracy

```
lr_weak_reg.score(X_test, y_test) → 0.86
lr_strong_reg.score (X_test, y_test → 0.88
```

~~regularized loss = ori~~

⊙ more regularization: (almost always) higher test accuracy.

## L1 vs L2 regularization:

⊙ Lasso = linear regression with L1 regularization

⊙ Ridge = linear " " L2 "

⊙ For other models like logistic regression we just say L1, L2 etc

```
lr_l1 = LogisticRegression(penalty='L1')
lr_l2 = " " ( ) # penalty='L2' by default
lr_l1.fit(X_train, y_train)
lr_l2.fit (X_train, y_train)
plt.plot (lr_l1.coef_.flatten()) → ~~can~~ also doing feature selection
plt.plot (lr_l2.coef_.flatten()) → shrinks coef to smaller.
```

## Model Coefficients for multiclass

```
# one-vs-rest by default.
lr_ovr = LogisticRegression()
lr_ovr.fit(X,y)
lr_ovr_coef_.shape
lr_ovr.intercept_.shape
```

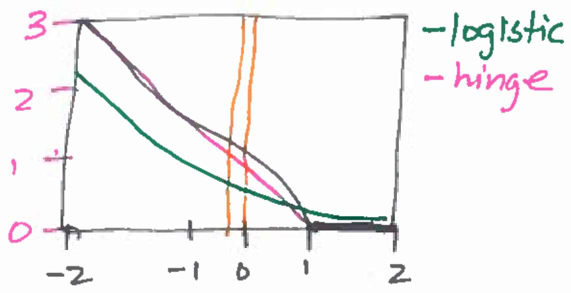
```
lr_mn = LogisticRegression(
        multi_class = "multinomial",
        solver = "lbfgs")
lr_mn.fit(X,y)
lr_mn.coef_.shape
lr_mn.~~coef_.s~~ intercept_.shape.
```

# 3-4 Support Vectors

May-6-2022

## What is an SVM?

- Linear ~~SVM~~ classifiers
- Trained using the hinge loss and L2 Regularization
- A key difference b/w hinge + logistic is in "flat" part of hinge loss, which occurs when the raw model output is greater than 1 → meaning you predicted an example correctly beyond some margin of error.



- If training example fall in this "zero loss" region, it doesn't contribute to the fit.
- If we remove that example, nothing would change.
- This the key property of SVMs
- Support vectors: a training example not in the flat part of the loss diagram.
- Support vector: an example that is incorrectly classified or close to the boundary.
- How close is considered close enough this is controlled by regularization strength.
- Support vectors are the examples that matter to your fit.
- Logistic regression doesn't has flat part so non-support vectors are also crucial in a way as for ~~LR~~ logReg all examples are crucial.
- Kernel SVM are fast to fit and predict.
- speed is determined by no-of support vectors rather than the whole training dataset.

## SGDClassifiers (Stochastic Gradient Descent)

- scales well to larger datasets.

```
from sklearn.linear_model import SGDClassifier
logreg = SGDClassifier(loss='log')

linsvm = SGDClassifier(loss='hinge')
```

- SGDClassifier hyperparameter alpha is like 1/C

# 4-1 Decision-Tree for Classification May-6-2022

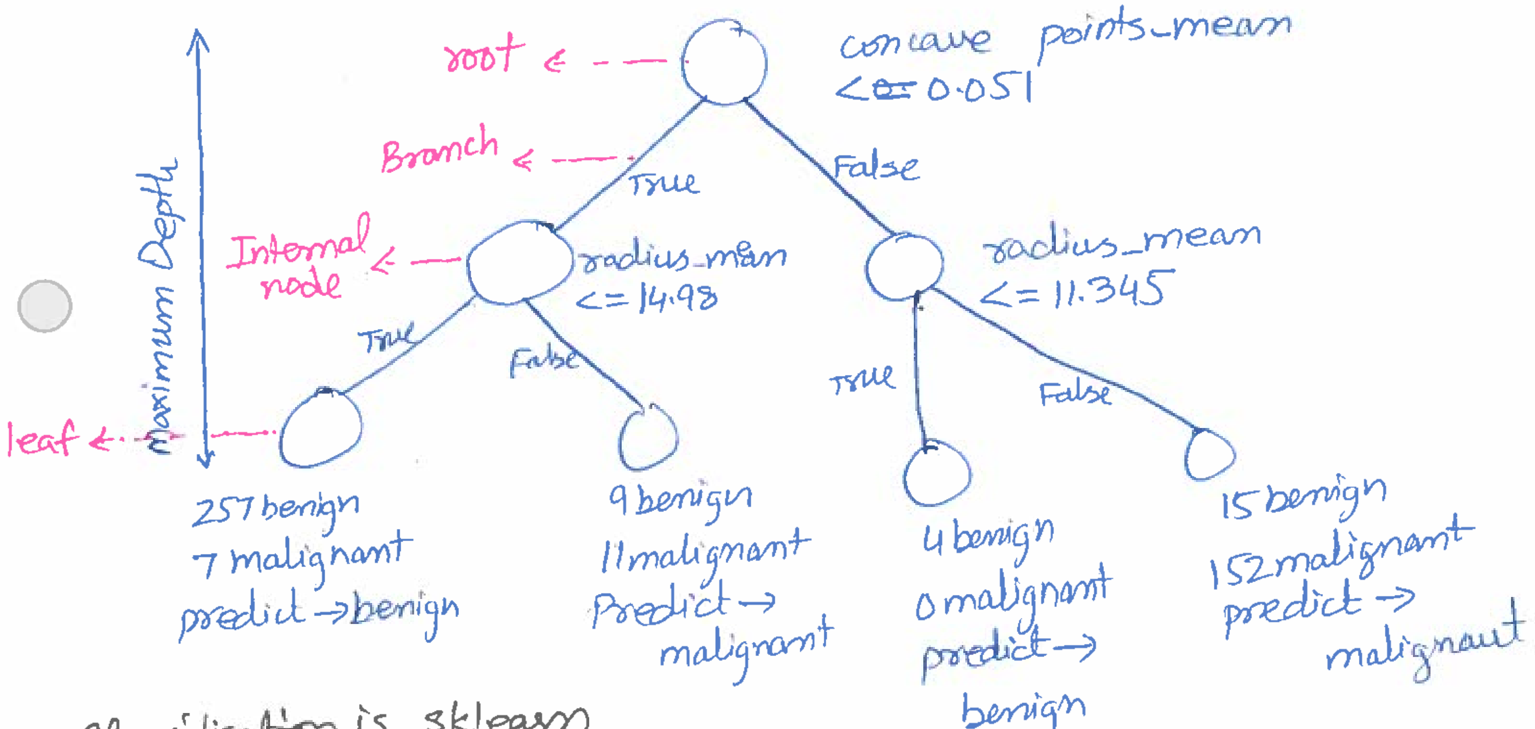
Classification and Regression Trees (CART)

## Classification-tree

- Sequence of if-else questions about individual features
- Objective: infer class labels
- Able to capture non-linear relationships b/w features and labels.
- Don't require features scaling (eg: Standardization,...)

## Breast Cancer Dataset in 2D

## Decision-tree diagram



## Classification is sklearn

```
# Import ~~decision tree~~ DecisionTreeClassifier
from sklearn.tree import DecisionTreeClassifier
# Import train_test_split
from sklearn.model_selection import train_test_split
# Import accuracy score
from sklearn.metrics import accuracy_score
# split data into 80% train, 20% test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    stratify=y,
                                                    random_state=1)
# Instantiate dt
dt = DecisionTreeClassifier(max_depth=2, random_state=1)
```

## Classification-Tree Learning

- Nodes are grown recursively
- At each ~~split~~ node, split the data based on:
  - feature f and split-point sp to maximize IG(node).
- If IG(node)=0, declare the node a leaf...

from sklearn.tree import DecisionTreeClassifier
" " .model-selection import train-test_split
" " .metrics import accuracy score.

X_train, X_test, y_train, y_test = train-test_split (X, y, test_size=0.2, stratify=y, random_state=1

dt=DecisionTreeClassifier (criterion='gini', random_state=1)

df.fit (X_train, y_train)

~~df~~ y_pred= dt.predict (X_test)

accuracy-score (y-test, y-pred)

---

## Decision-Tree for Regression

Auto-mpg Dataset → UCI ML Repo

Regression-Tree in sklearn

~~from~~ sklearn.tree import DecisionTree~~Classifier~~Regressor
" " .model_selection import train_test_split
" " .metrics import mean_squared_error as MSE

X_train, X-test, y-train, y-test = train_test_split (X, y, test_size=0.2, random_state=3)

dt= DecisionTree~~Classifier~~Regressor (max_depth=4, min_samples_leaf=0.1, random_state=3)

dt.fit (X_train, y_train)

y_pred= dt.predict (X_test)

mse_dt = MSE (y_test, y_pred)

rmse_dt = mse_dt**(1/2)

rmse_dt

# 4-2 Generalization Error

May-7-2022

## Difficulties

- Overfitting → fitting to noise in training set
- Underfitting → not fitting enough.

## Generalization Error

- does $\hat{f}$ generalize well on unseen data?
- It can be decomposed as follows:

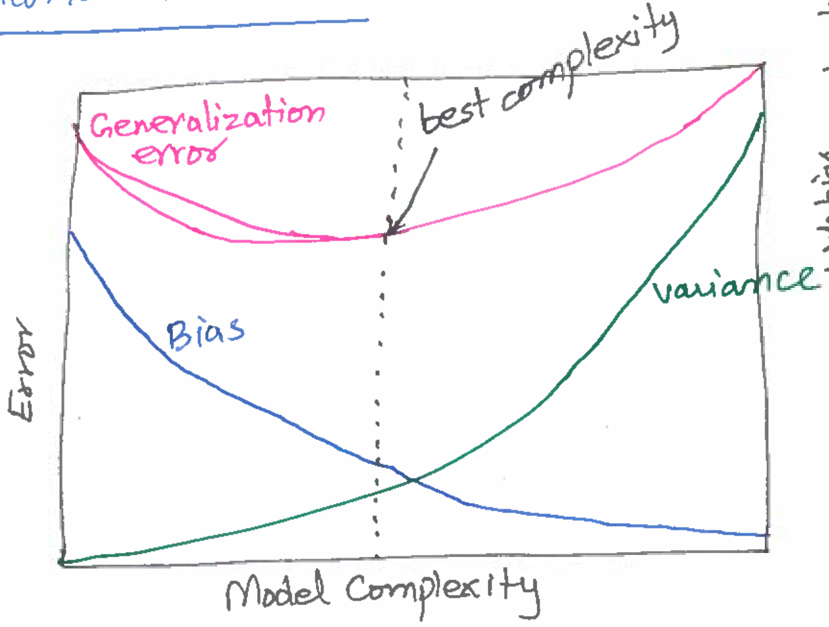
Generalization Error of $\hat{y}$ = bias$^2$ + variance + irreducible error

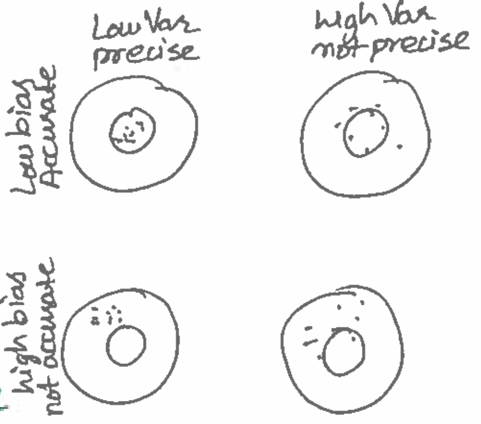
**Bias** → It tells you on average how much $\hat{f} \neq f$
→ high bias models lead to underfitting.

**Variance** → It tells you how much $\hat{f}$ is inconsistent over differen
training sets.
→ high variance models leads to overfitting.

**Model Complexity** → set the flexibility of $\hat{f}$
→ example → Maximum tree depth, minimum_samples_per_le

## Bias-Variance Tradeoff





```
print('CV MSE: {:.2f}'.format(MSE_CV.mean()))  →
  "  ('Train MSE: {:.2f}'. format (MSE(y_train, y_pred_train)))
  "  ('Test MSE:    "    "    "     (MSE(y_test, y_pred_test)))
```

## Ensemble Learning

### Advantages of CART

- Simple to understand
- simple to interpret
- easy to use.
- Flexibility: ability to describe non-linear dependencies.
- preprocessing: no need to standardize or normalize features.

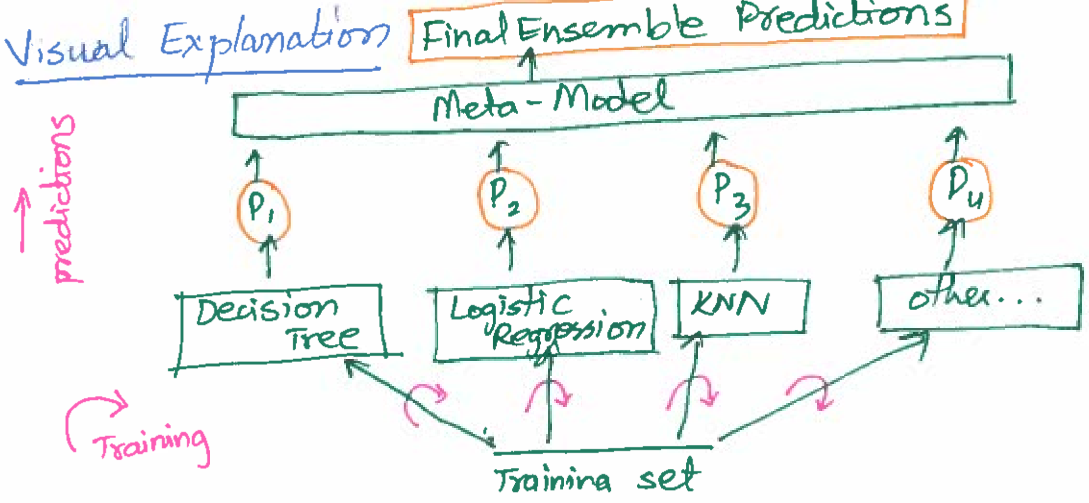
### Limitation of CART

- Classification: can only produce orthogonal decision boundaries.
- Sensitive to small variations in the training set.
- High Variance: unconstrained CARTs may overfit the training set.
- Solution: ensemble learning.

### Ensemble learning

- Train different models on the same dataset.
- let each model make its predictions.
- Meta-model: aggregates predictions of individual models.
- Final predictions: more robust and less prone to errors.
- Best results: models are skillful in different ways.

### Visual Explanation

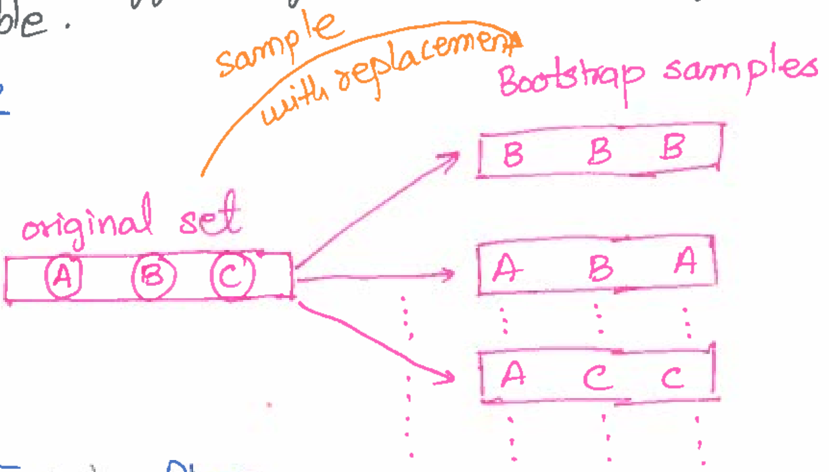


```
# Iterate over the defined list of tuples containing classifiers.
for clf_name , clf  in  classifiers:
      # fit clf to training set
        clf.fit (X_train, y_train)

      # Predict the labels of test set
        y_pred = clf. predict (X_test)

      # Evaluate accuracy of clf on test set
      print ('{:s} : {:.3f}. format (clf_name, accuracy_score(y_tes
                                                              y_pred
# Instantiate a VotingClassifier 'vc'
  vc = VotingClassifier(estimators = classifiers)
# fit 'vc' to training set and predict test set labels

 vc. fit(X_train, y_train)

 y_pred = vc. predict (X_test)
# Evaluate the test accuracy of 'vc'
 ~~pring~~ print('Voting Classifier: {.3f}'.formal(accuracy_score(
                                                y_test, y_pred))
```

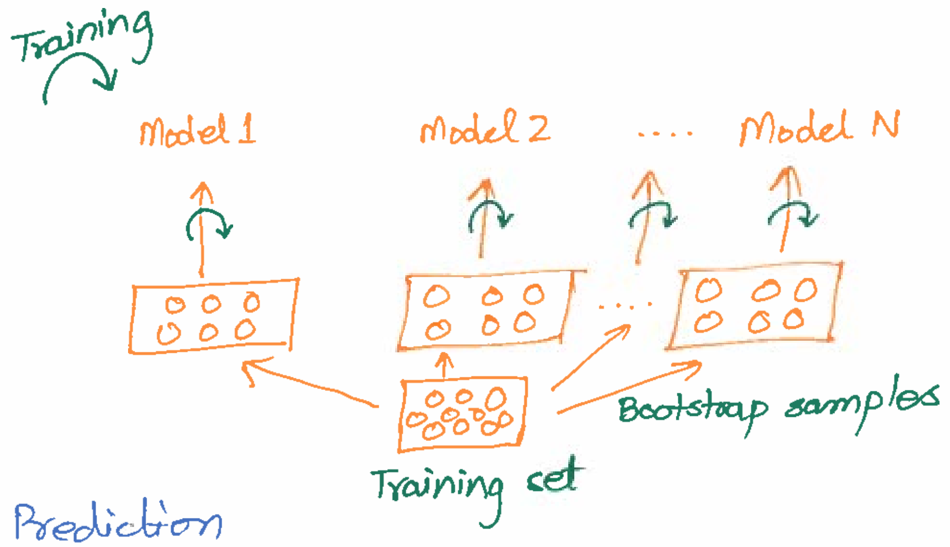
# 4-3 Bagging May-8-2022

4.3.1 An ensemble method → Bootstrap Aggregating (Bagging)

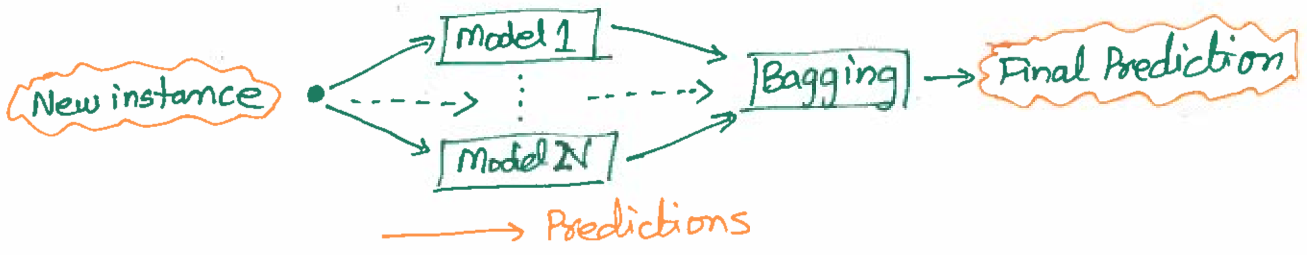
- Voting Classifier
  - same training set } final prediction by majority voting
  - ≠ algorithms.
- Bagging
  - one algorithm,
  - ≠ subsets of the training set } - each model is trained on subset of traini data
  - uses a technique ≐ Bootstrap
  - It has the effect of reducing variance of individual models in the ensemble.
- Bootstrap



- Bagging: Training Phase



Bagging : Prediction



4.3.2
## Out of Bag Evaluation

- Bagging

⊙ Some instances may be sampled several times for one model

⊙ other instances may not be sampled at all.

- Out of Bag (OOB) instances :

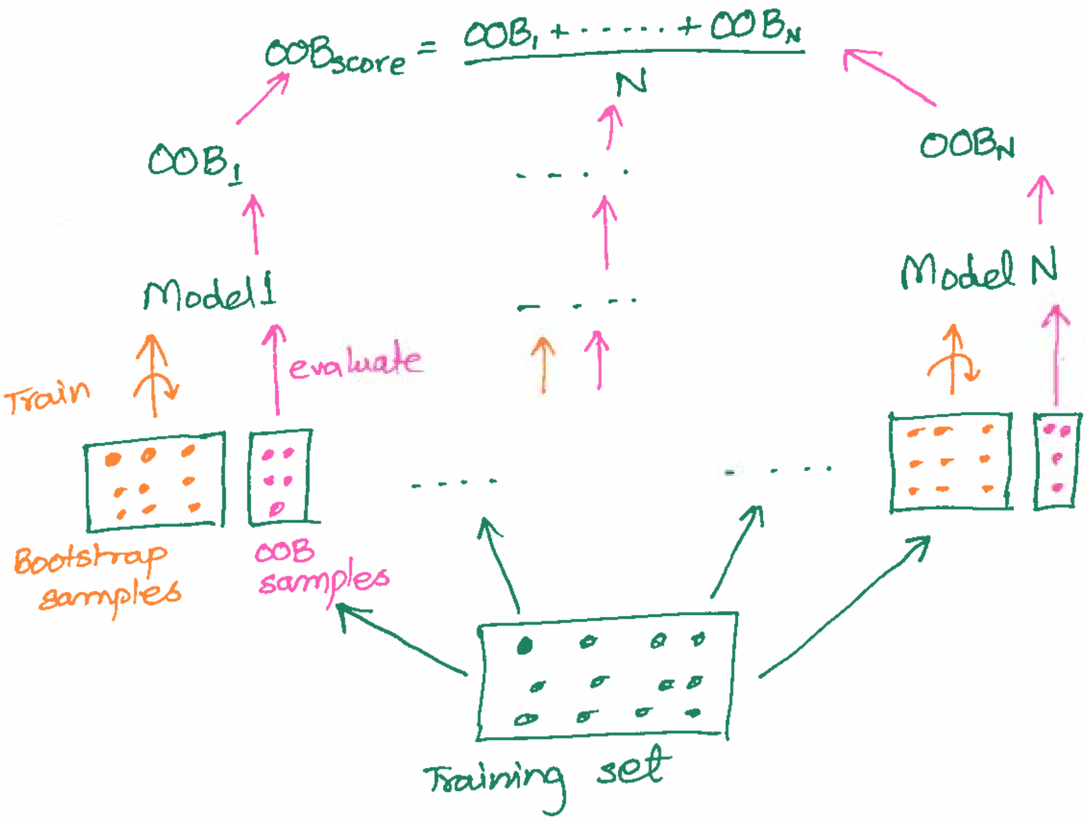
⊙ On average, ~~63%~~ for each model, 63% of training instances are sampled.

⊙ The remaining 37% constitute the OOB instances.

⊙ Since OOB instances are not seen by the model during trainin
these can be used to estimate the performance of the ensemble
without the need for crossvalidation

⊙ This technique is called OOB-evaluation.

- OOB - Evaluation



## 4.3.3 Random Forests (ensemble learning method)

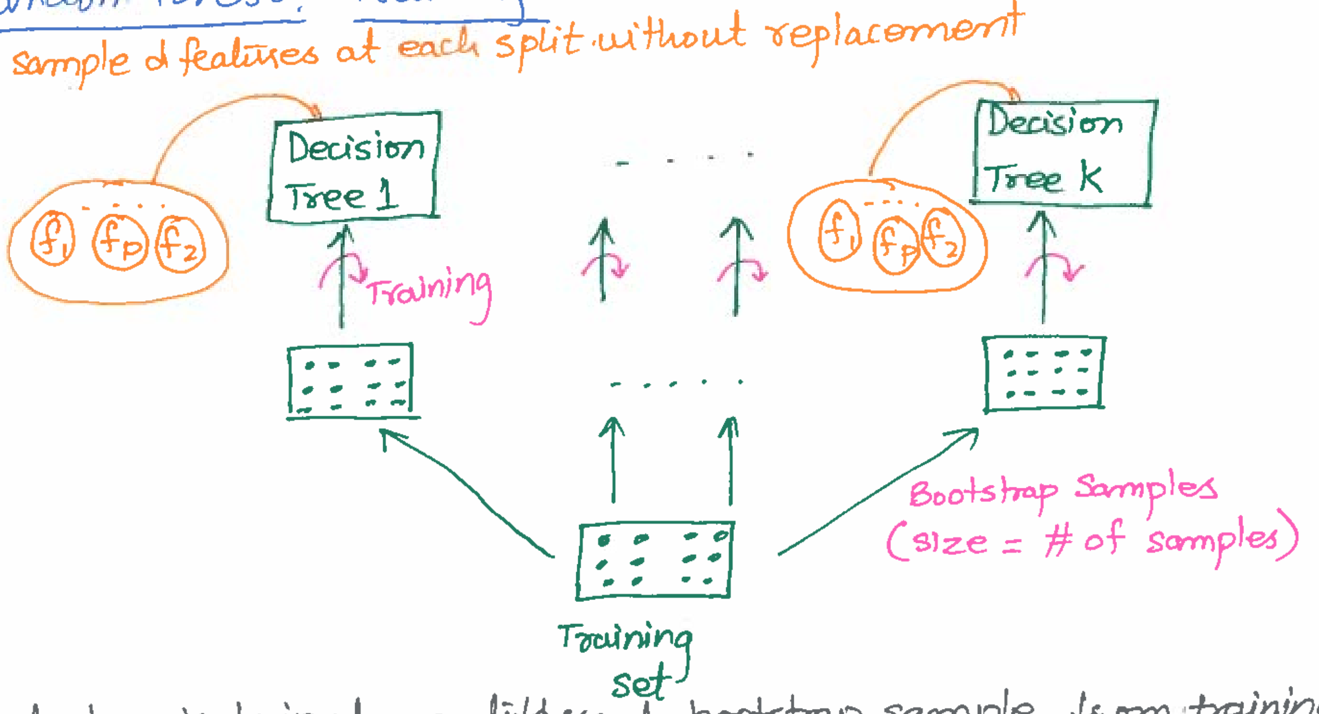
### Bagging

- each estimator can be any model: DecisionTree, Logistic Regression, Neural Net, ....
- Each estimator is trained on a distinct bootstrap samples of training set
- estimators use all available features for training and prediction.

### Random Forest

- Base estimator: Decision Tree
- Each estimator is trained on a different bootstrap sample having the same size as training set.
- RF introduces further randomization in the training of individual trees.
- d features are sampled at each node without replacement.
  (d < total number of features)

### Random Forest: Training



- Each tree is trained on different bootstrap sample from training set
- In addition, when a tree is trained, at each node, only d features are sampled from all features without replacement
- The node is then split using the sampled feature that minimizes info gain
- In sklearn default value of d is $\sqrt{\# \text{ of features}}$

## Feature Importance

- Tree based methods: enable measuring the importance of each feature in prediction.
- In sklearn:
  - how much the tree node use a particular feature (weighted avg.) to reduce impurity.
  - can be accessed using the attribute feature_importance_

## Feature Importance in sklearn

- To visualize the importance of features as assessed by rf. We can create a pandas series of feature importances as shown ~~here~~ below and then sort this series and make a horizontal barplot.

```
import pandas as pd
import matplotlib.pyplot at plt
importances_rf = pd.Series(rf.feature_Importances_, index=X.columns)
sorted_importances_rf = importances_rf.sort_values()
sorted_   ,,      ,,   .plot (kind="barh", color='lightgreen')
plt.show()
```

## 44.1 Ada Boosting

Boosting

- Refers to ensemble method in which many predictors are trained and each predictor learns from the errors of its predecessor.
- ensemble method combining several weak learners to form a strong learner.
- Weak learner: Model doing slightly better than guessing.
- ~~Exp~~ Eg: Decision stump (CART whose maximum depth is 1)

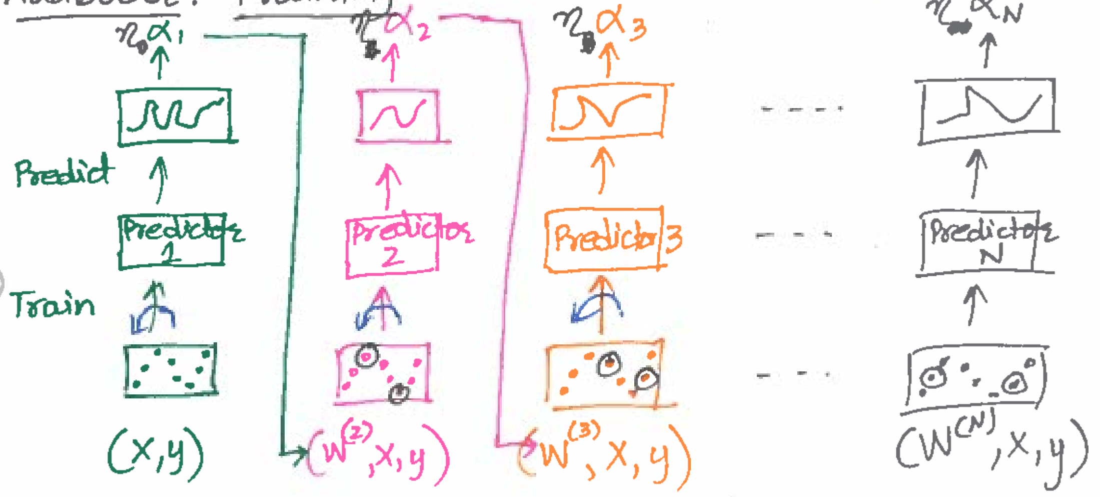
Boosting

- Train an ensemble of predictors sequentially.
- Each predictor tries to correct its predecessor.
- Most popular boosting methods:
  - ⊙ Ada Boost.
  - ⊙ Gradient Boosting.

AdaBoost

- Stands for adaptive boosting
- Each predictor pays more attention to the instances wrongly predicted by its predecessor.
- Acheived by changing the weights of training instances.
- Each predictor is assigned a coefficient $\alpha$.
- $\alpha$ depends on the predictor's training error.

AdaBoost: Training

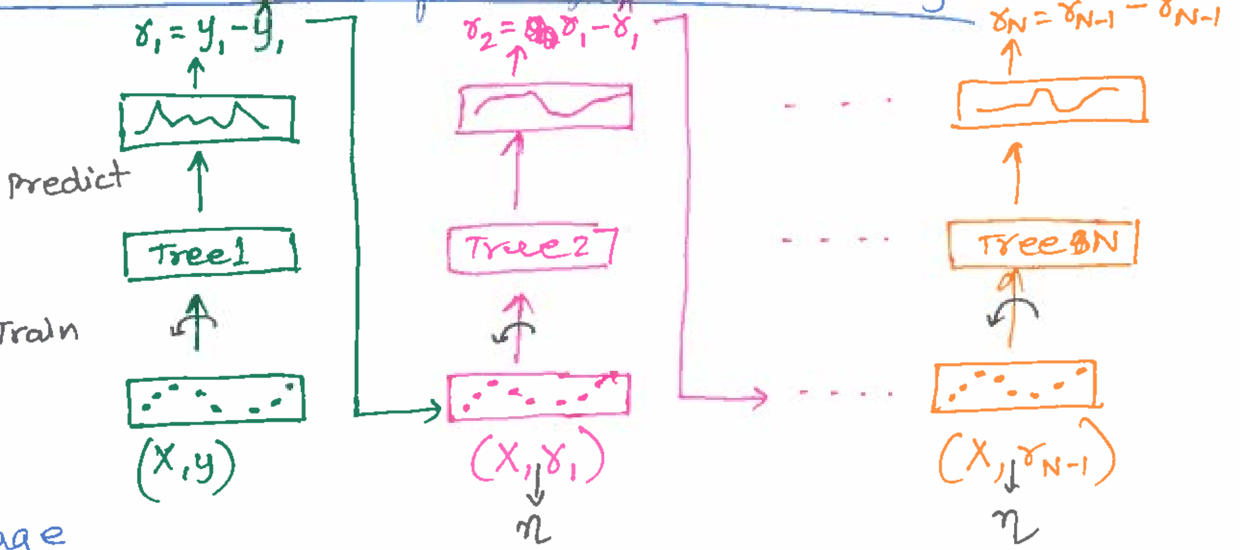


$\eta$ = learning rate (0 – 1)

## 4.4.2 Gradient Boosting (GB)

- Sequential correction of predecessor's errors.
- Does not tweak the weights of training instances like AdaBoost
- Fit each predictor is trained using its predecessor's residual errors as labels.
- Gradient Boosted trees: a CART is used as a base learner.

### Gradient Boosted trees for Regression: Training.



### Shrinkage

- learning rate ($\eta$) (0-1)
- prediction of each ~~tree~~ tree in the ensemble is shrinked after it is multiplied by learning rate ($\eta$)
- Just like AdaBoost there is a trade off b/w eta and no-of estimators
- $\downarrow$ eta $\rightarrow \uparrow$ no-of estimators.

### Gradient Boosted Trees: ~~Reg~~ Prediction

#### Regression:

- $y_{pred} = y_1 + \eta r_1 + \cdots + \eta r_N$
- In sklearn: Gradient Bosting Regressor.

#### Classification

- In sklearn: Gradient Boosting Classifier.

## 4.4.3 Stochastic Gradient Boosting (SGB)

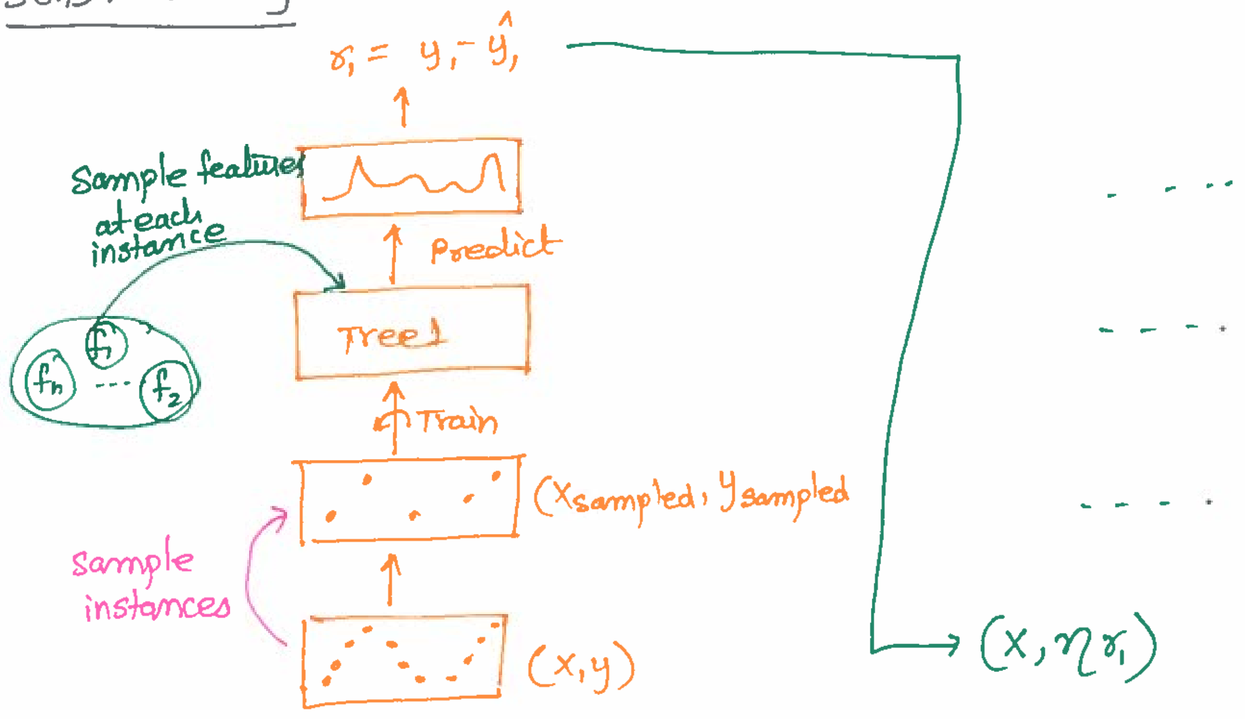
### Cons of GB:

- GB involves exhaustive search procedure
- Each CART is trained to find the best split points and features.
- May lead to CARTs using the same split points and maybe the same features.

### Stochastic GB:

- Each tree is trained on a random subset of rows of the training data.
- The sampled instances (40%-80% of training set) are sampled without replacement.
- Features are sampled (without replacement) when choosing split points.
- Result: further ensemble diversity.
- Effect: adding further variance to the ensemble of trees.

### SGB: Training



## 4.5.1 Tuning a CART's Hyperparameters

- To ~~ob~~ obtain better performance hyperparameters of ML model should be tuned.
- ML models are characterized by parameters and hyperparameters.
  - ⊙ parameters: learn from data
    - CART example: split-point of a node, split-feature of a node, ....
  - ⊙ hyperparameter: not learned from data, ~~a~~ set prior to training.
    - CART example: max_depth, min_samples_leaf, splitting criterion, ......etc.

Hyperparameter tuning

* Problem: search for a set of optimal hyperparameters for a learning algorithm.

* Solution: find a set of optimal hyperparameters that results in an optimal model.

* Optimal Model: yeilds and optimal score.

* Score: in sklearn defaults to accuracy (classification) and $R^2$ (regression)

* Cross-Validation: is used to estimate the generalization performance

Why tune Hyperparameters?

- In sklearn - a model's ~~hypesp~~ default hyperparameters are not optimal for all problems.
- Hyperparameters should be tuned to obtain the best model performance.

Approaches to hyperparameter tuning

⊙ Grid Search

⊙ Randomized Search

⊙ Bayesian Optimization

⊙ Genetic Algorithms.

⊙ . . . . . .

→

```
# Extract best hyperparameters from 'grid_dt'
best_~~param~~hyperparams = grid_dt.best_params_
print ('Best hyperparameters : \n', best_hyperparams)
# Extract best CV score from 'grid_dt'
best_CV_score = grid_dt.best_score_
print ('Best CV accuracy:'. format(best_CV_score))
# Extract best model from grid_dt
best_model = grid_dt.best_estimator_
# Evaluate test set accuracy
test_acc = best_model.score (X_test, y_test)
print ('Test set accuracy of best model: {:.3f}'. format(test_~~score~~acc))
```

## 4.5.2 Tuning an RF's Hyperparameters

**Random Forest hyperparameters**

- ~~Car~~' CART hyperparameters
- no. of estimators
- bootstrap
- . . . . .

**Tuning is expensive:**

- computationally expensive
- sometimes leads to very slight improvement

(*) For above reasons it is desired to weight the impact of tuning on the pipeline of your data analysis project as a whole in order to understand if it is worth pursuing.

**Inspecting RF hyperparameters in sklearn**

```
from sklearn.ensemble import RandomForestRegressor
SEED = 1
rf = RandomForestRegressor(random_state = SEED)
# inspect rf's hyperparameters.

 rf.get_params()
```

**Autodataset**

```
from sklearn.metrics import mean_squared_error as MSE
from sklearn.model_selection import GridSearchCV

params_rf = { 'n_estimators': [300, 400, 500],
              'max_depth' : [4, 6, 8],
              'min_samples_leaf' : [0.1, 0.2],
              'max_features': ['log2', 'sqrt'] }
                                 param_grid = param_rf
grid_rf = GridSearchCV(estimator = rf ^ cv= 3, scoring= "neg_mean_squared_err
                                  verbose = 1, n_jobs = -1)

grid_rf.fit(X_train, y_train)
best_hyperparams = grid_rf.best_params_
print ('Best hyperparameters :\n', best_hyperparams)
```

datacamp

# Python For Data Science Scikit-Learn Cheat Sheet

Learn Scikit-Learn online at www.DataCamp.com

## Scikit-learn

Scikit-learn is an open source Python library that implements a range of machine learning, preprocessing, cross-validation and visualization algorithms using a unified interface.

### A Basic Example

```
>>> from sklearn import neighbors, datasets, preprocessing
>>> from sklearn.model_selection import train_test_split
>>> from sklearn.metrics import accuracy_score
>>> iris = datasets.load_iris()
>>> X, y = iris.data[:, :2], iris.target
>>> X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=33)
>>> scaler = preprocessing.StandardScaler().fit(X_train)
>>> X_train = scaler.transform(X_train)
>>> X_test = scaler.transform(X_test)
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
>>> knn.fit(X_train, y_train)
>>> y_pred = knn.predict(X_test)
>>> accuracy_score(y_test, y_pred)
```

## > Loading The Data

Also see NumPy & Pandas

**Your data needs to be numeric and stored as NumPy arrays or SciPy sparse matrices. Other types that are convertible to numeric arrays, such as Pandas DataFrame, are also acceptable.**

```
>>> import numpy as np
>>> X = np.random.random((10,5))
>>> y = np.array(['M','M','F','F','M','F','M','M','F','F','F'])
>>> X[X < 0.7] = 0
```

## > Training And Test Data

```
>>> from sklearn.model_selection import train_test_split
>>> X_train, X_test, y_train, y_test = train_test_split(X,
                                                        y,
                                                        random_state=0)
```

## > Model Fitting

**Supervised learning**

```
>>> lr.fit(X, y) #Fit the model to the data
>>> knn.fit(X_train, y_train)
>>> svc.fit(X_train, y_train)
```

**Unsupervised Learning**

```
>>> k_means.fit(X_train) #Fit the model to the data
>>> pca_model = pca.fit_transform(X_train) #Fit to data, then transform it
```

## > Prediction

**Supervised Estimators**

```
>>> y_pred = svc.predict(np.random.random((2,5))) #Predict labels
>>> y_pred = lr.predict(X_test) #Predict labels
>>> y_pred = knn.predict_proba(X_test) #Estimate probability of a label
```

**Unsupervised Estimators**

```
>>> y_pred = k_means.predict(X_test) #Predict labels in clustering algos
```

## > Preprocessing The Data

### Standardization

```
>>> from sklearn.preprocessing import StandardScaler
>>> scaler = StandardScaler().fit(X_train)
>>> standardized_X = scaler.transform(X_train)
>>> standardized_X_test = scaler.transform(X_test)
```

### Normalization

```
>>> from sklearn.preprocessing import Normalizer
>>> scaler = Normalizer().fit(X_train)
>>> normalized_X = scaler.transform(X_train)
>>> normalized_X_test = scaler.transform(X_test)
```

### Binarization

```
>>> from sklearn.preprocessing import Binarizer
>>> binarizer = Binarizer(threshold=0.0).fit(X)
>>> binary_X = binarizer.transform(X)
```

### Encoding Categorical Features

```
>>> from sklearn.preprocessing import LabelEncoder
>>> enc = LabelEncoder()
>>> y = enc.fit_transform(y)
```

### Imputing Missing Values

```
>>> from sklearn.preprocessing import Imputer
>>> imp = Imputer(missing_values=0, strategy='mean', axis=0)
>>> imp.fit_transform(X_train)
```

### Generating Polynomial Features

```
>>> from sklearn.preprocessing import PolynomialFeatures
>>> poly = PolynomialFeatures(5)
>>> poly.fit_transform(X)
```

## > Create Your Model

### Supervised Learning Estimators

**Linear Regression**

```
>>> from sklearn.linear_model import LinearRegression
>>> lr = LinearRegression(normalize=True)
```

**Support Vector Machines (SVM)**

```
>>> from sklearn.svm import SVC
>>> svc = SVC(kernel='linear')
```

**Naive Bayes**

```
>>> from sklearn.naive_bayes import GaussianNB
>>> gnb = GaussianNB()
```

**KNN**

```
>>> from sklearn import neighbors
>>> knn = neighbors.KNeighborsClassifier(n_neighbors=5)
```

### Unsupervised Learning Estimators

**Principal Component Analysis (PCA)**

```
>>> from sklearn.decomposition import PCA
>>> pca = PCA(n_components=0.95)
```

**K Means**

```
>>> from sklearn.cluster import KMeans
>>> k_means = KMeans(n_clusters=3, random_state=0)
```

## > Evaluate Your Model's Performance

### Classification Metrics

**Accuracy Score**

```
>>> knn.score(X_test, y_test) #Estimator score method
>>> from sklearn.metrics import accuracy_score #Metric scoring functions
>>> accuracy_score(y_test, y_pred)
```

**Classification Report**

```
>>> from sklearn.metrics import classification_report #Precision, recall, f1-score and support
>>> print(classification_report(y_test, y_pred))
```

**Confusion Matrix**

```
>>> from sklearn.metrics import confusion_matrix
>>> print(confusion_matrix(y_test, y_pred))
```

### Regression Metrics

**Mean Absolute Error**

```
>>> from sklearn.metrics import mean_absolute_error
>>> y_true = [3, -0.5, 2]
>>> mean_absolute_error(y_true, y_pred)
```

**Mean Squared Error**

```
>>> from sklearn.metrics import mean_squared_error
>>> mean_squared_error(y_test, y_pred)
```

**R² Score**

```
>>> from sklearn.metrics import r2_score
>>> r2_score(y_true, y_pred)
```

### Clustering Metrics

**Adjusted Rand Index**

```
>>> from sklearn.metrics import adjusted_rand_score
>>> adjusted_rand_score(y_true, y_pred)
```

**Homogeneity**

```
>>> from sklearn.metrics import homogeneity_score
>>> homogeneity_score(y_true, y_pred)
```

**V-measure**

```
>>> from sklearn.metrics import v_measure_score
>>> metrics.v_measure_score(y_true, y_pred)
```

### Cross-Validation

```
>>> from sklearn.cross_validation import cross_val_score
>>> print(cross_val_score(knn, X_train, y_train, cv=4))
>>> print(cross_val_score(lr, X, y, cv=2))
```

## > Tune Your Model

### Grid Search

```
>>> from sklearn.grid_search import GridSearchCV
>>> params = {"n_neighbors": np.arange(1,3),
              "metric": ["euclidean", "cityblock"]}
>>> grid = GridSearchCV(estimator=knn,
                        param_grid=params)
>>> grid.fit(X_train, y_train)
>>> print(grid.best_score_)
>>> print(grid.best_estimator_.n_neighbors)
```

### Randomized Parameter Optimization

```
>>> from sklearn.grid_search import RandomizedSearchCV
>>> params = {"n_neighbors": range(1,5), "weights": ["uniform", "distance"]}
>>> rsearch = RandomizedSearchCV(estimator=knn, param_distributions=params,
                                 cv=4, n_iter=8, random_state=5)
>>> rsearch.fit(X_train, y_train)
>>> print(rsearch.best_score_)
```

Learn Data Skills Online at www.DataCamp.com