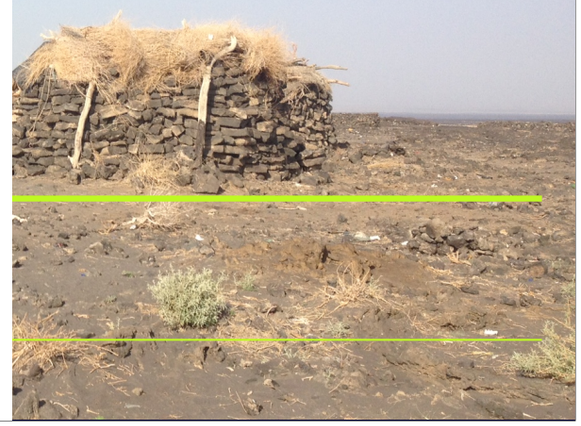

MODELS, EDA AND EVERYTHING THAT BETWEEN:
MACHINE LEARNING SUM
DOR MIZRAHI


SUBJECTS
-BASIC MATH.
-EDA.
-SEBORN N MATPLOTLIB.
-CLASSIFICATION VS REGRESSION.
-K FOLD/cross validation
-RANDOM SEARCH VS GRID SEARCH.
-CONFUSION MATRIX.
-ROC.
-ENSEMBLE METHODES.-(bagging, pasting, voting, stacking)
DOR MIZRAHI

SUBJECTS:

MODELS:

-KNN
-LINEAR REGRESSION
-LOGISTIC REGRRESSION
-SVM
-NAIVE BAYES
-DECISION TREE
-RANDOM FOREST
-EXTRA TREE
-XG BOOST
-LGBM
-CATBOOST
-K MEANS



BASIC MATH:

4 basic ways to merge distance-

1,2. Minkowski n Euclidian- are the same. The shortest distance between two points is a straight line. MINKOVSKI is for more then 2 dimensions.

3. Manhattan-distance between two vectors if they could only move right angles.

4.Cosine similetary- The cosine similarity is simply the cosine of the angle between two vectors.
Two vectors with exactly the same orientation have a cosine similarity of 1.

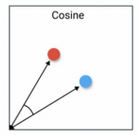
$$D(x,y) = \left(\sum_{i=1}^{n} |x_i - y_i|^p\right)^{\frac{1}{p}}$$

2. Euclidean Distance

$$D(x,y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2)}$$

3. Manhattan Distance

$$D(x,y) = \sum_{i=1}^{k} |x_i - y_i|$$



$$D(x,y) = cos(\theta) = \frac{x \cdot y}{\|x\| \, \|y\|}$$

EDA-Scaling

1) *Min Max Scaler* $x_{new} = \frac{x - x_{min}}{x_{max} - x_{min}}$

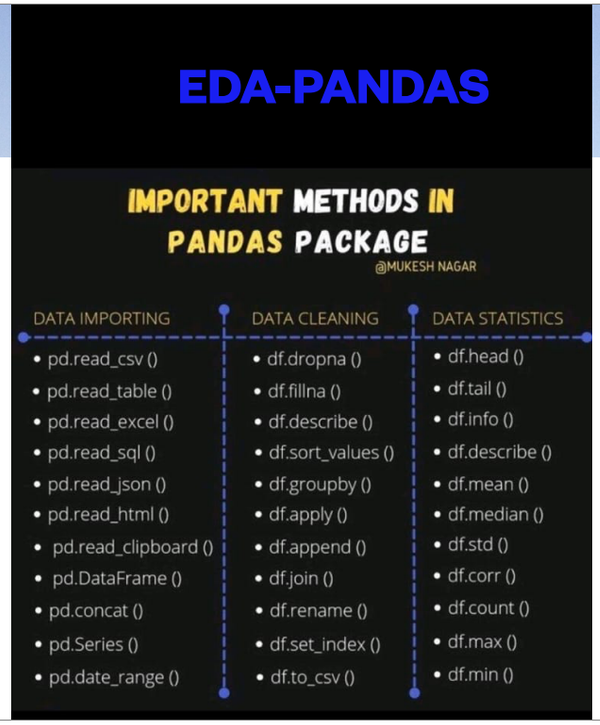
2) *Standard Scaler* $x_{new} = \frac{x - \mu}{\sigma}$ *s normally distributed within each feature and scales them such that the distribution centered around 0, with a standard deviation of 1.*

3) *Max Abs Scaler-Scale each feature by its maximum absolute value. This estimator scales and translates each feature individually such that the maximal absolute value of each feature in the training set is 1.0*

4) *Robust Scaler*
5) Quantile Transformer Scaler
6) Power Transformer Scaler
7) Unit Vector Scale

https://towardsdatascience.com/all-about-feature-scaling-bcc0ad75cb35

EDA-PANDAS


IMPORTANT METHODS IN PANDAS PACKAGE
@MUKESH NAGAR
DATA IMPORTING
• pd.read_csv ()
• pd.read_table ()
• pd.read_excel ()
• pd.read_sql ()
• pd.read_json ()
• pd.read_html ()
• pd.read_clipboard ()
• pd.DataFrame ()
• pd.concat ()
• pd.Series ()
• pd.date_range ()
DATA CLEANING
• df.dropna ()
• df.fillna ()
• df.describe ()
• df.sort_values ()
• df.groupby ()
• df.apply ()
• df.append ()
• df.join ()
• df.rename ()
• df.set_index ()
• df.to_csv ()
DATA STATISTICS
• df.head ()
• df.tail ()
• df.info ()
• df.describe ()
• df.mean ()
• df.median ()
• df.std ()
• df.corr ()
• df.count ()
• df.max ()
• df.min ()

EDA

Examples:

```
import pandas as pd
import numpy as np
import matplotlib as mp
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import scipy.stats as st
from sklearn import ensemble, tree, linear_model
import missingno as msno
#pip install missingno
```

Importing

```
df = pd.read_csv('./employees.csv')
df.head()
df.shape
```

Importing df

```
df.describe()
df.info()
df.isnull().sum()
```

Checking

```
print("Before",df.shape)
df = df.dropna( how = 'any')
print("After",df.shape)
print(df.isnull().sum())
```

Deleting nulls

```
PD.rename(columns={'status':'parkinson'}, inplace=True)
```

Rename

Droping

```
PD.drop('name',axis=1,inplace=True)
```

```
df['BMI'].nunique()
```

MATPLOTLIB

CODE Examples:

Making subplot for all column

```
plt.figure(figsize=(20,30))
i = 1
for column in PD.columns:
    plt.subplot(6,4,i)
    plt.hist(PD[column], color='green', edgecolor = 'black', alpha = 1/2)
    plt.title(column)
    i += 1
```

```
#pairplot of the data
sns.pairplot(df_kyphosis,
hue="Kyphosis")
plt.show()
```

KYPHOSIS IS NAME

```
#plot confusion matrix
#I send him the Classifier and the X test and the Y test
#and I also tell him to give me in the graph only integers
#and in addition I tell him to show the graph in shades of blue
plot_confusion_matrix(knn1, X_test, y_test,values_format="d",cmap='Blues')
#Perform a prediction on the Classifier and give it the X test as a parameter
```

Heat map

```
import plotly.express as px
plt.figure(figsize=(20,10))
sns.heatmap(PD.corr(),
annot=True,cmap='Blues')
```

MATPLOTLIB

CODE Examples:

```
def plot_svm_boundary(model,X,y):

    X = X.values
    y = y.values

    # Scatter Plot
    plt.scatter(X[:, 0], X[:, 1], c=y, s=30,cmap='seismic')


    # plot the decision function
    ax = plt.gca()
    xlim = ax.get_xlim()
    ylim = ax.get_ylim()

    # create grid to evaluate model
    xx = np.linspace(xlim[0], xlim[1], 30)
    yy = np.linspace(ylim[0], ylim[1], 30)
    YY, XX = np.meshgrid(yy, xx)
    xy = np.vstack([XX.ravel(), YY.ravel()]).T
    Z = model.decision_function(xy).reshape(XX.shape)

    # plot decision boundary and margins
    ax.contour(XX, YY, Z, colors='k', levels=[-1, 0, 1], alpha=0.5,
               linestyles=['--', '-', '--'])
    # plot support vectors
    ax.scatter(model.support_vectors_[:, 0], model.support_vectors_[:, 1], s=100,
               linewidth=1, facecolors='none', edgecolors='k')
    plt.show()
```

Results graphs

```
X_ax=range(len(y_test))
plt.figure(figsize=(12, 6))
plt.plot(X_ax, y_test, label="original")
plt.plot(X_ax, y_prediction, label="predicted")
plt.title("Car price")
plt.xlabel('X')
plt.ylabel('Price')
plt.legend(loc='best',fancybox=True, shadow=True)
plt.grid(True)
plt.show()
```

CLASSIFCATION VS REGRASSION

CLASSIFICATION -

when we want to class from **2** different groups. Examples: 1/0, **YES OR NO**.

REGRESSION-

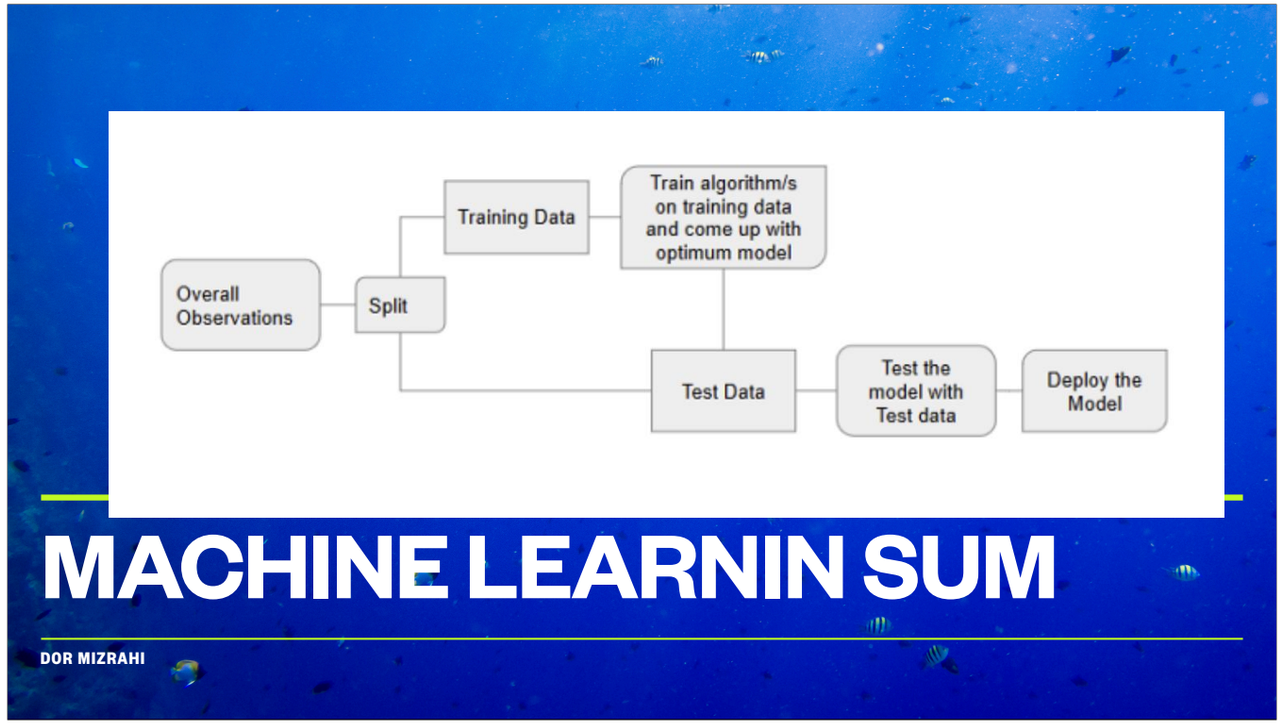
is when we want to get numeric answer. Examples: price of a product, temperature, grade..

* in both there is an $x_{(n)}$ features and a y label we want to predict.

CLASSIFCATION VS REGRASSION

 # for almost every model there is a classifier and a regressor.

Algorithm	Task
Linear Regression	Regression
Logistic Regression	Classification
K-Nearest Neighbors (KNN)	Regression, Classification
Decision Trees	Regression, Classification
Support Vector Machine (SVM)	Regression, Classification
Artificial Neural Network (ANN)	Regression, Classification
K-Means	Clustering



MACHINE LEARNIN SUM

DOR MIZRAHI

K-fold code example:

```
from sklearn.datasets import load_iris
from sklearn.model_selection import cross_val_score,KFold
from sklearn.linear_model import LogisticRegression
iris=load_iris()
X=iris.data
Y=iris.target
logreg=LogisticRegression()
kf=KFold(n_splits=5)
score=cross_val_score(logreg,X,Y,cv=kf)
print("Cross Validation Scores are {}".format(score))
print("Average Cross Validation score :{}".format(score.mean()))
```

CROSS VALIDATION-K FOLD:

-problem: when we don't have a enough data, splitting could cost for not fitting our model enough.
-K-fold- answers that problem. goal of cross validation is to validate the model several times without scarifying data available to train the model.

CROSS VALIDATION-K FOLD:

Pros:

1. The whole dataset is used as both a training set and validation set:

Cons:

1. Not to be used for imbalanced datasets: As discussed in the case of HoldOut cross-validation, in the case of K-Fold validation too it may happen that all samples of training set will have no sample form class “1” and only of class “0”.And the validation set will have a sample of class “1”.

2. Not suitable for Time Series data: For Time Series data the order of the samples matter. But in K-Fold Cross-Validation, samples are selected in random order.

GRID SEARCH:

CODE EXAMPLE:

```
>>> from sklearn import svm, datasets
>>> from sklearn.model_selection import GridSearchCV
>>> iris = datasets.load_iris()
>>> parameters = {'kernel':('linear', 'rbf'), 'C':[1, 10]}
>>> svc = svm.SVC()
>>> clf = GridSearchCV(svc, parameters)
>>> clf.fit(iris.data, iris.target)
>>> sorted(clf.cv_results_.keys())
['mean_fit_time', 'mean_score_time', 'mean_test_score',...
 'param_C', 'param_kernel', 'params',...
 'rank_test_score', 'split0_test_score',...
 'split2_test_score', ...
 'std_fit_time', 'std_score_time', 'std_test_score']
```

A DICTIONARY OF THE PARAMS WE WANT TO TRY.

THE MODEL

GRID SEARCH:

known as an exhaustive search, Grid Search looks through each combination of hyperparameters.
This means that every combination of specified hyperparameter values will be tried.
Significant Params:

n_jobs*int, default=None*
Number of jobs to run in parallel. `None` means 1 unless in a `joblib.parallel_backend` context. `-1` means using all processors. See Glossary for more details. refitbool, str, or callable, default=True

Refit an estimator using the best found parameters on the whole dataset.

cv*int, cross-validation generator or an iterable, default=None*
Determines the cross-validation splitting strategy.

Verbose:int- ***how much you want him to write on process..***
return_train_score*bool, default=False*
If `False`, the `cv_results_` attribute will not include training scores

RANDOM SEARCH:-CODE EXAMPLE

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegression
>>> from sklearn.model_selection import RandomizedSearchCV
>>> from scipy.stats import uniform
>>> iris = load_iris()
>>> logistic = LogisticRegression(solver='saga', tol=1e-2, max_iter=200,
...                               random_state=0)
>>> distributions = dict(C=uniform(loc=0, scale=4),
...                      penalty=['l2', 'l1'])
>>> clf = RandomizedSearchCV(logistic, distributions, random_state=0)
>>> search = clf.fit(iris.data, iris.target)
>>> search.best_params_
{'C': 2..., 'penalty': 'l1'}
```

RANDOM SEARCH:

In contrast to GridSearchCV, not all parameter values are tried out, but rather a fixed number of parameter settings is sampled from the specified distributions. The number of parameter settings that are tried is given by n_iter.

*If all parameters are presented as a list, sampling without replacement is performed. If at least one parameter is given as a distribution, sampling with replacement is used. It is highly recommended to use continuous distributions for continuous parameters.

Considerable params:

estimator*estimator object*
A object of that type is instantiated for each grid point. This is assumed to implement the scikit-learn estimator interface. Either estimator needs to provide a `score` function, or `scoring` must be passed.

n_iter*int, default=10*
Number of parameter settings that are sampled. n_iter trades off runtime vs quality of the solution.

random_state*int, RandomState instance or None, default=None*
Pseudo random number generator state used for random uniform sampling from lists of possible values

KNearestNeighborsClassifier-CODEXAMPLE

```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 3, metric = 'minkowski', p = 2)
classifier.fit(X_train, y_train)
```

```
for i in range (10):
    param_grid={'n_neighbors':[3,5,7,9] , 'metric':['euclidean','manhattan'], 'p':[1,2,3]}
    knn_classifior = KNeighborsClassifier()
    grid_search = GridSearchCV(knn_classifior, param_grid, cv=5, scoring='accuracy')
    grid_search.fit(X_train, y_train)
    print(grid_search.best_params_)
    bestscore=grid_search.best_score_
```

KNN-KNeighborsClassifier

PARAMETERS:

n_neighbors*int, default=5*

Number of neighbors to use by default for `kneighbors` queries.

weights*{'uniform', 'distance'} or callable, default='uniform'*

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights

algorithm*{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*

Algorithm used to compute the nearest neighbors:

- 'ball_tree' will use `BallTree`
- 'kd_tree' will use `KDTree`
- 'brute' will use a brute-force search.
- 'auto' will attempt to decide the most appropriate algorithm based on the values passed to `fit` method.

leaf_size*int, default=30*

Leaf size passed to BallTree or KDTree. This can affect the speed of the construction and query, as well as the memory required to store the tree. The optimal value depends on the nature of the problem

p:*int, default=2*

Power parameter for the Minkowski metric. When p = 1, this is equivalent to using manhattan_distance (l1), and euclidean_distance (l2) for p = 2. For arbitrary p, minkowski_distance (l_p) is used.

metric:*str or callable, default='minkowski'*

Metric to use for distance computation. Default is "minkowski", which results in the standard Euclidean distance when p = 2

KNN-KNeighborsClassifier

Classification is computed from a simple majority vote of the nearest neighbors of each point: a query point is assigned the data class which has the most representatives within the

nearest neighbors of the point.

The optimal choice of the value k is highly data-dependent: in general a larger k suppresses the effects of noise, but makes the classification boundaries less distinct.

KNN-KNearestNeighbors-codexamples:

```
 from sklearn.neighbors import KNeighborsRegressor
>>> neigh = KNeighborsRegressor(n_neighbors=2)
```

class sklearn.neighbors.**KNeighborsRegressor**(*n_neighbors=5*, ***, *weights='uniform'*, *algorithm='auto'*, *leaf_size=30*, *p=2*, *metric='minkowski'*, *metric_params=None*, *n_jobs=None*

```
max_accur = 0.0
for k in range (3,20,2):
  knn1 = KNeighborsClassifier(n_neighbors=k, weights='uniform')
  knn1.fit(X_train,y_train)
  accur = knn1.score(X_test,y_test)
  if accur > max_accur:
    max_accur = accur
    best_k = k
```

KNN-KNeighborsREGRESSOR-codexamples:

The target is predicted by local interpolation of the targets associated of the nearest neighbors in the training set.

Params

weights*{'uniform', 'distance'} or callable, default='uniform'*

Weight function used in prediction. Possible values:

- 'uniform' : uniform weights. All points in each neighborhood are weighted equally.
- 'distance' : weight points by the inverse of their distance. in this case, closer neighbors of a query point will have a greater influence than neighbors which are further away.
- [callable] : a user-defined function which accepts an array of distances, and returns an array of the same shape containing the weights.

Uniform weights are used by default.

algorithm:*{'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'*

p:*int, default=2*

Power parameter for the Minkowski metric

LINEAR REGRESSION-CODE EXAMPLES:


```
from sklearn.linear_model import LinearRegression
regressor = LinearRegression()
regressor.fit(X_train, y_train)
```

LINEAR REGRESSION:

This model tries to fit a straight line to all points

Params:

fit_intercept: ***bool, default=True***

Whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations.

normalize*bool, default=False*

Use ur scaler instead

positive:*bool, default=False*

When set to `True`, forces the coefficients to be positive. This option is only supported for dense arrays.

****See also**

`Ridge`

Ridge regression addresses some of the problems of Ordinary Least Squares by imposing a penalty on the size of the coefficients with l2 regularization.

`Lasso`

The Lasso is a linear model that estimates sparse coefficients with l1 regularization.

`ElasticNet`

Elastic-Net is a linear regression model trained with both l1 and l2 -norm regularization of the coefficients.

Logistic REGRESSION:

A linear that classifies. What's on lablel 1 what's on 0. In the middle it uses probability of one event (out of two alternatives) taking place by having the log-odds.

penalty*{'l1', 'l2', 'elasticnet', 'none'}, default='l2'*
Specify the norm of the penalty:

- `'none'`: no penalty is added;
- `'l2'`: add a L2 penalty term and it is the default choice;
- `'l1'`: add a L1 penalty term;
- `'elasticnet'`: both L1 and L2 penalty terms are added.

dual:*bool, default=False*
Dual or primal formulation. Dual formulation is only implemented for l2 penalty with liblinear solver. Prefer dual=False when n_samples > n_features.

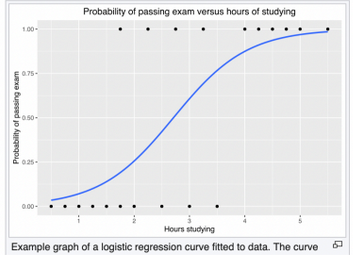
C:*float, default=1.0*
Inverse of regularization strength; must be a positive float. Like in support vector machines, smaller values specify stronger regularization.

fit_intercept:*tbool, default=True*
Specifies if a constant (a.k.a. bias or intercept) should be added to the decision function.

class_weight*dict or 'balanced', default=None*
Weights associated with classes in the form `{class_label: weight}`. If not given, all classes are supposed to have weight one.

solver*{'newton-cg', 'lbfgs', 'liblinear', 'sag', 'saga'}, default='lbfgs'*
Algorithm to use in the optimization problem. Default is 'lbfgs'. To choose a solver, you might want to consider the following aspects:

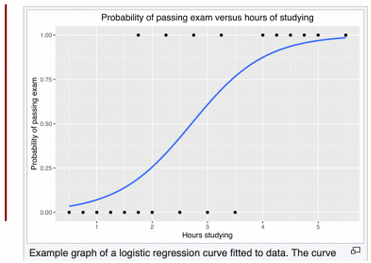
- For small datasets, 'liblinear' is a good choice, whereas 'sag' and 'saga' are faster for large ones;
- For multiclass problems, only 'newton-cg', 'sag', 'saga' and 'lbfgs' handle multinomial loss;
- 'liblinear' is limited to one-versus-rest schemes.



Example graph of a logistic regression curve fitted to data. The curve

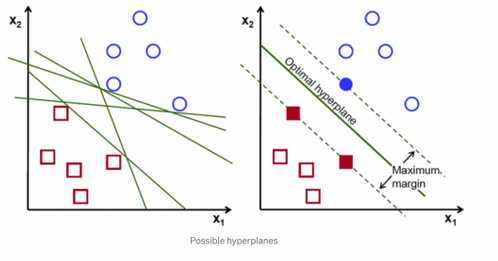
Logistic REGRESSION:

```
>>> from sklearn.datasets import load_iris
>>> from sklearn.linear_model import LogisticRegression
>>> X, y = load_iris(return_X_y=True)
>>> clf = LogisticRegression(random_state=0).fit(X, y)
```



Example graph of a logistic regression curve fitted to data. The curve

SVM:support vector machine



Support vectors are data points that are closer to the hyperplane and influence the position and orientation of the hyperplane. Using these support vectors, we maximize the margin of the classifier. Deleting the support vectors will change the position of the hyperplane. These are the points that help us build our SVM.

SVM:support vector machine

C:*float, default=1.0*
Regularization parameter. The strength of the regularization is inversely proportional to C. Must be strictly positive. The penalty is a squared l2 penalty.

kernel:*{'linear', 'poly', 'rbf', 'sigmoid', 'precomputed'} or callable, default='rbf'*
Specifies the kernel type to be used in the algorithm. If none is given, 'rbf' will be used. If a callable is given it is used to pre-compute the kernel matrix from data matrices; that matrix should be an array of shape

degree:*int, default=3*
Degree of the polynomial kernel function ('poly'). Ignored by all other kernels.

decision_function_shape*{'ovo', 'ovr'}, default='ovr'*
Whether to return a one-vs-rest ('ovr') decision function of shape (n_samples, n_classes) as all other classifiers, or the original one-vs-one ('ovo') decision function of libsvm which has shape (n_samples, n_classes * (n_classes - 1) / 2). However, note that internally, one-vs-one ('ovo') is always used as a multi-class strategy to train models; an ovr matrix is only constructed from the ovo matrix. The parameter is ignored for binary classification.

SVM:support vector machine-examples:

Classifier:

```
>>> import numpy as np
>>> from sklearn.pipeline import make_pipeline
>>> from sklearn.preprocessing import StandardScaler
>>> X = np.array([[-1, -1], [-2, -1], [1, 1], [2, 1]])
>>> y = np.array([1, 1, 2, 2])
>>> from sklearn.svm import SVC
>>> clf = make_pipeline(StandardScaler(), SVC(gamma='auto'))
```

Regressor:

```
svr_rbf = SVR(kernel="rbf", C=100, gamma=0.1, epsilon=0.1)
svr_lin = SVR(kernel="linear", C=100, gamma="auto")
svr_poly = SVR(kernel="poly", C=100, gamma="auto", degree=3, epsilon=0.1, coef0=1)
```

NAIVE BAYES:

Naïve Bayes is a classification technique based on **Bayes' Theorem**.
Let's assume that you are data scientist working major bank in NYC and you want to classify a new client as **eligible to retire or not**.
Customer **features** are his/her **age** and **salary**
*These features might be dependant on each others, however, we assume they are all independent and that's why its 'Naive'!

NAÏVE BAYES: SOME MATH!

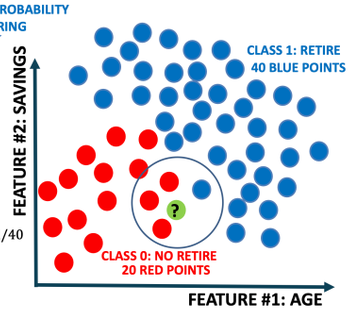
LIKELIHOOD

PRIOR PROBABILITY OF RETIRING

$$P(Retire|X) = \frac{P(X|Retire) * P(Retire)}{P(X)}$$

MARGINAL LIKELIHOOD

- $P(Retire) = \frac{\#\text{ of Retiring}}{\text{Total points}} = 40/60$
- $P(X|Retire) = \frac{\#\text{ of smilar observations for retiring}}{Total\ \#\ retiring} = 1/40$
- $P(X) = \frac{\#\text{ of Similar observations}}{Total\ \#\ Points} = 4/60$
- $P(Retire|X) = \frac{\frac{40}{60} * \frac{1}{40}}{\frac{4}{60}} = \frac{1/60}{4/60} = 0.25$



NAIVE BAYES:

Sagnificant parameters:

priors:*array-like of shape (n_classes,)*
Prior probabilities of the classes. If specified, the priors are not adjusted according to the data.

var_smoothing:*float, default=1e-9*
Portion of the largest variance of all features that is added to variances for calculation stability.

See also:

`BernoulliNB`
Naive Bayes classifier for multivariate Bernoulli models.
`CategoricalNB`
Naive Bayes classifier for categorical features.
`ComplementNB`
Complement Naive Bayes classifier.
`MultinomialNB`
Naive Bayes classifier for multinomial models.

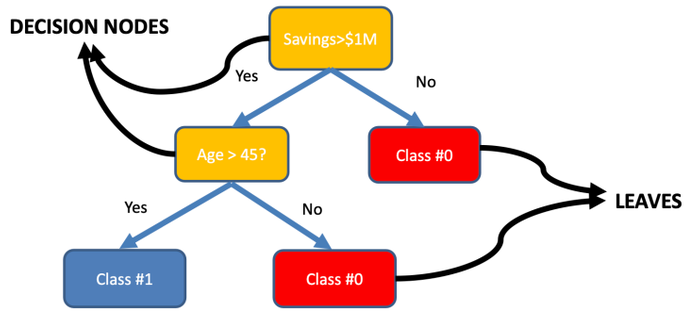
NAIVE BAYES:code example

```
>>> clf = GaussianNB()
>>> clf.fit(X, Y)
GaussianNB()
```

priors : array-like of shape (n_classes,)
    Prior probabilities of the classes. If specified, the priors are not adjusted according to the data.
var_smoothing : float, default=1e-9
    Portion of the largest variance of all features that is added to variances for calculation stability.

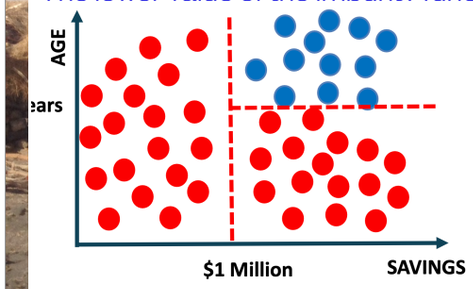
DECISION TREE:

Decision Trees are supervised Machine Learning technique where the data is split according to a certain condition/parameter. The tree consists of **decision nodes** and **leaves**.
Leaves are the decisions or the final outcomes. Decision nodes are where the data is split based on a certain attribute. Objective is to minimize the entropy which provides the optimum split



DECISION TREE:

Impurity measurement
-Two most common impurity functions are Entropy and Gini (index).
-the range of Entropy is from 0 to 1
- and the range of Gini Index is from 0 to 0.5 -The higher value of the impurity function either Entropy or Gini, the more impure the node is.
-The lower value of the impurity function, the more pure the node is .



DECISION TREE:

THE MOST SIGNIFICANT THING HERE AND IN RANDOM FOREST IS THE **MAX DEPTH.**

If the MAX DEPTH will be too high there will be OVERFITTING.

If the MAX DEPTH will be too low there will be UNDERFITTING.

DECISION TREE:-params:

criterion:*{“gini”, “entropy”, “log_loss”}, default=”gini”*
The function to measure the quality of a split. Supported criteria are “gini” for the Gini impurity and “log_loss” and “entropy” both for the Shannon information gain

splitter:*{“best”, “random”}, default=”best”*
The strategy used to choose the split at each node. Supported strategies are “best” to choose the best split and “random” to choose the best random split.

max_depth:*int, default=None*
The maximum depth of the tree. If None, then nodes are expanded until all leaves are pure or until all leaves contain less than min_samples_split samples

min_samples_split:*int or float, default=2*
The minimum number of samples required to split an internal node:

min_samples_leaf:*int or float, default=1*
The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches.

min_weight_fraction_leaf:*float, default=0.0*
The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

class_weight:*dict, list of dict or “balanced”, default=None*
Weights associated with classes in the form `{class_label: weight}`. If None, all classes are supposed to have weight one. For multi-output problems, a list of dicts can be provided in the same order as the columns of y.

DECISION TREE:-CODE EXAMPLES:

```
from sklearn.tree import DecisionTreeRegressor
dtmodule = DecisionTreeRegressor(max_depth=10, random_state=99)
dtmodule.fit(X_train,y_train)
```

REGRESSOR

```
>>> from sklearn.model_selection import cross_val_score
>>> from sklearn.tree import DecisionTreeClassifier
>>> clf = DecisionTreeClassifier(random_state=0)
>>> iris = load_iris()
>>> cross_val_score(clf, iris.data, iris.target, cv=10)
```

CLASSIFIER

CLASSIFIER WITH MOST PARAMS

class sklearn.tree.**DecisionTreeClassifier**(*, *criterion='gini'*, *splitter='best'*, *max_depth=None*, *min_samples_split=2*, *min_samples_leaf=1*, *min_weight_fraction_leaf=0.0*, *max_features=None*, *random_state=None*, *max_leaf_nodes=None*, *min_impurity_decrease=0.0*, *class_weight=None*, *ccp_alpha=0.0*)

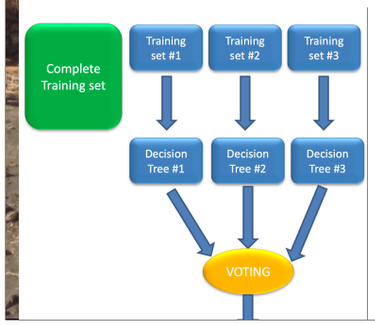
RANDOM FOREST:

It overcomes the issues with single decision trees by reducing the effect of noise.
Overcomes overfitting problem by taking average of all the predictions, canceling out biases.
Suppose training set: [X1, X2, X3, X4] with labels: [L1, L2, L3, L4]

Random forest creates three decision trees taking inputs as follows:

- **[X1, X2, X3], [X1, X2, X4], [X2, X3, X4]**

Example:
Combining votes from a pool of experts, each will bring their own experience and background to solve the problem resulting in a better outcome.



RANDOM FOREST REGRESSOR:

PARAMETERS:

n_estimators*:int, default=100*
The number of trees in the forest.

criterion:*{"squared_error", "absolute_error", "poisson"}, default="squared_error"*
The function to measure the quality of a split

max_depth:*int, default=None*
The maximum depth of the tree.

min_samples_split:*int or float, default=2*
The minimum number of samples required to split

min_samples_leaf:*int or float, default=1*
The minimum number of samples required to be at a leaf node.

min_weight_fraction_leaf:*float, default=0.0*
The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node. Samples have equal weight when sample_weight is not provided.

max_features:*{"sqrt", "log2", None}, int or float, default=1.0*

bootstrap:*bool, default=True*
Whether bootstrap samples are used when building trees. If False, the whole dataset is used to build each tree.

oob_score: *bool, default=False*
Whether to use out-of-bag samples to estimate the generalization score. Only available if bootstrap=True.

n_jobs:*int, default=None*
The number of jobs to run in parallel. `fit`, `predict`, `decision_path` and `apply` are all parallelized over the trees.

ccp_alpha :*non-negative float, default=0.0*
Complexity parameter used for Minimal Cost-Complexity Pruning. The subtree with the largest cost complexity that is smaller than `ccp_alpha` will be chosen

max_samples:*int or float, default=None*
If bootstrap is True, the number of samples to draw from X to train each base estimator.

RANDOM FOREST classifier:

```
randomforest_params={'n_estimators':[10,20,30,40,50,60,70,80,90,100],
                     'max_depth':[2,3,4,5,6,7,8,9,10],
                     'criterion':['gini','entropy'],
                     'random_state':[0]}
randomforest_classifier = RandomForestClassifier()#max features is set to auto
grid_search = GridSearchCV(randomforest_classifier, param_grid=randomforest_params, cv=5, n_jobs=-1, verbose=1)


from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()
parameters = {'n_estimators':[50,100,150,200], 'max_depth':[6,7,8,9,10]}
grid_search = GridSearchCV(rf, parameters, cv=10)
grid_search.fit(X_train, y_train)

grid_search.best_score_
```

RANDOM FOREST EXAMPLES:

```
>>> regr = RandomForestRegressor(max_depth=2, random_state=0)
>>> regr.fit(X, y)

RandomForestClassifier(bootstrap=True, class_weight=None, criterion='entropy',
            max_depth=None, max_features='auto', max_leaf_nodes=None,
            min_impurity_decrease=0.0, min_impurity_split=None,
            min_samples_leaf=1, min_samples_split=2,
            min_weight_fraction_leaf=0.0, n_estimators=100, n_jobs=1,
            oob_score=False, random_state=None, verbose=0,
            warm_start=False)
```


ENSEMBLE METHODS

Ensemble methods

The goal of ensemble methods is to combine the predictions of several base estimators built with a given learning algorithm in order to improve generalizability / robustness over a single estimator

Bias/ Variance Tradeoff

Methods that minimize variance:

Bagging

Random forests

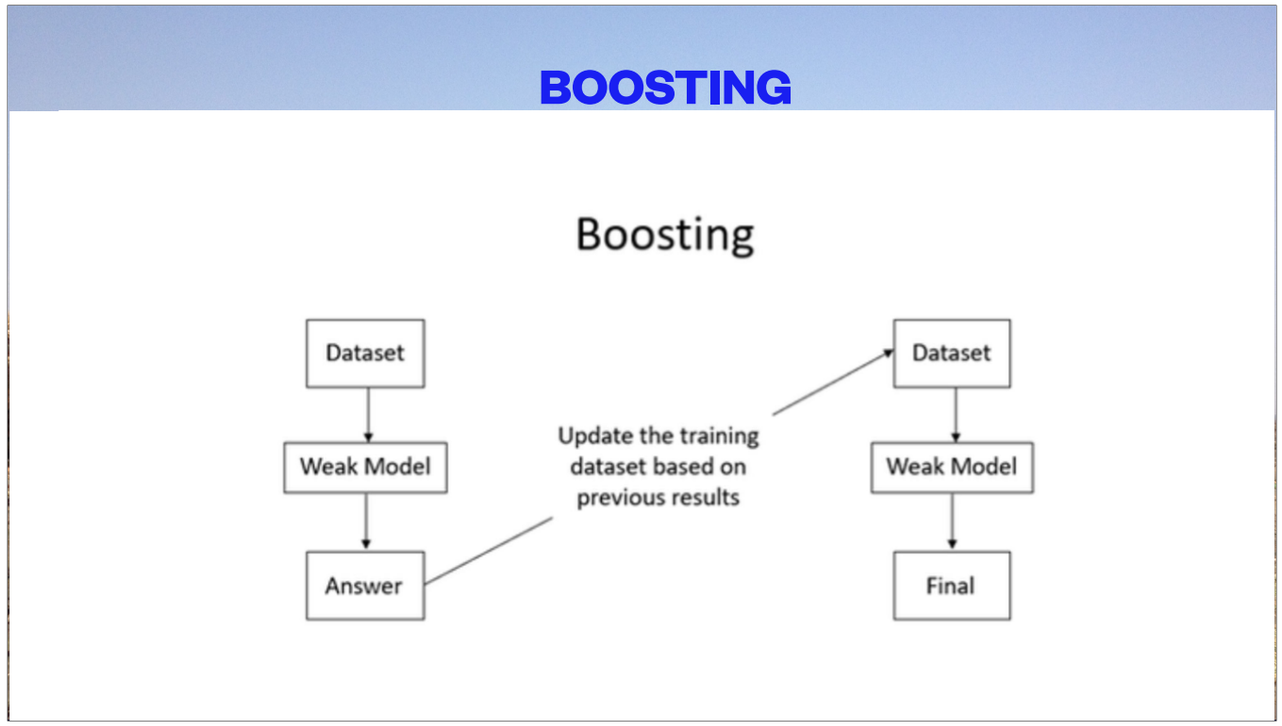
Ensamble methods that minimize bias:

Gradient descent

Boosting

Ensemble selection

BOOSTING


Boosting
Dataset
Weak Model
Answer
Update the training dataset based on previous results
Dataset
Weak Model
Final

Boosting

The main idea of boosting is to sequentially combine many weak models (a model performing slightly better than random chance) and thus through greedy search create a strong competitive predictive model.

BOOSTING-ada

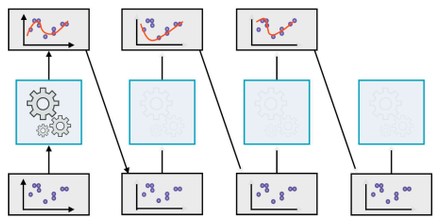
~~Ada~~ptive ~~Boost~~ing (AdaBoost)

Increase relative weights of those instances that the first predictor got wrong

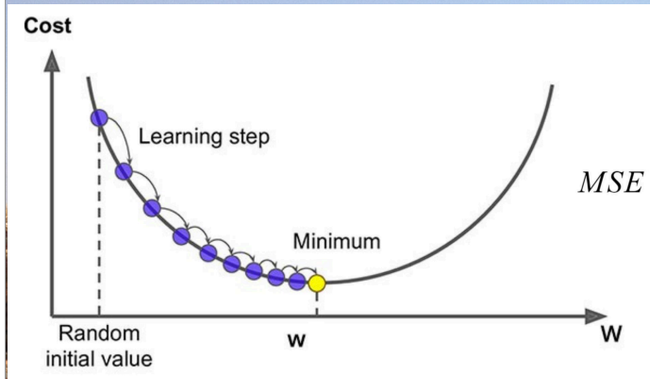
Construct and train models sequentially

Misclassified training points from each model are up-weighted

Next model sees previously misclassified points more often



BOOSTING-gradient



BOOSTING-gradient

Gradient Boosting

Model 1:

$$y = A_1 + B_1x + e_1$$

Model 2:

$$y = A_2 + B_2x + e_2$$

Model 3:

$$y = A_3 + B_3x + e_3$$

"Strong learner" when combined

Combined Model:

$$y = A_1 + A_2 + A_3 + (B_1 + B_2 + B_3)\,x + e$$

BOOSTING-code

```
ada_reg = AdaBoostRegressor(DecisionTreeRegressor(max_depth=4),
                            n_estimators=100,
                            learning_rate=1.0)#<--------learning rate-----------

ada_reg.fit(x_train, y_train)
ada_reg.estimator_weights_


gbr = GradientBoostingRegressor(max_depth=3, n_estimators=3, learning_rate=1.0)

gbr.fit(x_train, y_train)


gbr = GradientBoostingRegressor(max_depth=3, n_estimators=30, learning_rate=0.1)
```

BAGGING

bootstrapping refers to a resample method
- consists of repeatedly drawn, with replacement
- samples from data to form other smaller datasets - Like making a bunch of simulations to our original dataset
- so in some cases we can generalize the mean and the standard deviation.
- each bootstrap sample containing n observations

Drawn data with replacement: the observations can appear more than one time in a single sample
Bagging means bootstrap + aggregating
- it is a ensemble method
- in which we first bootstrap our data
- and for each bootstrap sample we train one model. - After that, we aggregate them with equal weights.

Pasting

bootstrapping refers to a resample method
- consists of repeatedly drawn, with replacement
- samples from data to form other smaller datasets - Like making a bunch of simulations to our original dataset
- so in some cases we can generalize the mean and the standard deviation.
- each bootstrap sample containing n observations

Drawn data with replacement: the observations can appear more than one time in a single sample
Bagging means bootstrap + aggregating
- it is a ensemble method
- in which we first bootstrap our data
- and for each bootstrap sample we train one model. - After that, we aggregate them with equal weights.

-When it's not used replacement, the method is called pasting.

Bagging or Pasting?

-Since pasting is without replacement,
- each subset of the sample can be used once at most,
- which means that you need a big dataset for it to work.
-pasting was originally designed for large data-sets, when computing power is limited.
-Bagging, on the other hand, can use the same subsets many times, which is great for smaller sample sizes, in which it improves robustness (to my experience).
- Thus the size is the major factor for making this decision. - If your sample size is small, pasting isn't a real option.

-The following
-trains an ensemble of 500 decision tree classifiers (n_estimators),
-each trained on 100 training instances randomly sampled from the training set
-with replacement (bootstrap=True).
-If you want to use pasting we simply set bootstrap=False instead.

OOB-Out Of BAGGING SCORING

Out-of-Bag Scoring

-If we are using bagging, there's a chance that a sample would never be selected, while anothers may be selected multiple time.

-The probability of not selecting a specific sample is (1–1/n), where n is the number of samples.

-Therefore, the probability of not picking n samples in n draws is (1–1/n)^n.

- When the value of n is big, we can approximate this probability to 1/e,

- which is approximately 0.3678.

- This means that when the dataset is big enough, 37% of its samples are never selected and we could use it to test our model.

This is called Out-of-Bag scoring, or OOB Scoring.

Bagging Tips:

Bagging generally gives better results than Pasting

- - Good results come around the 25% to 50% row sampling mark
- - Random patches and subspaces should be used while dealing with high dimensional data

- To find the correct hyperparameter values we can do GridSearchCV / RandomSearchCV

BAGGING

```
from sklearn.ensemble import BaggingClassifier, BaggingRegressor
bag = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),#<-------------------------------
    n_estimators=500,
    max_samples=0.5,
    bootstrap=True,
    random_state=42
)

bag = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    n_estimators=500,
    max_samples=1.0,
    bootstrap=False,  #<-------------------this-----------------
    max_features=0.5,  #<-------------------------------
    bootstrap_features=True,#<-------------------------------
    random_state=42
)

bag = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    n_estimators=500,
    max_samples=0.25,
    bootstrap=True,#<-------------------this-----------------
    max_features=0.5,
    bootstrap_features=True,#<-------------------this-----------------
    random_state=42
)

bag = BaggingClassifier(
    base_estimator=DecisionTreeClassifier(),
    n_estimators=500,
    max_samples=0.25,
    bootstrap=True,
    oob_score=True,#<-------------------------------
    random_state=42
)

from sklearn.ensemble import BaggingRegressor

bag_regressor = BaggingRegressor(random_state=1)
bag_regressor.fit(X_train, Y_train)
```

from sklearn.ensemble import BaggingClassifier from sklearn.tree import DecisionTreeClassifier bagging_clf = BaggingClassifier(DecisionTreeClassifier(max_leaf_nodes=20), n_estimators=500, max_samples=100, bootstrap=True, n_jobs=-1)

Bootstrap

Random Subspaces (JUST subsets of columns)

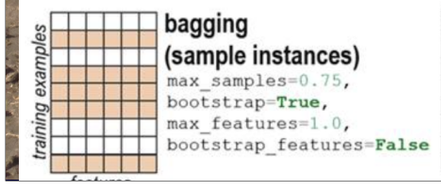
bag = BaggingClassifier(base_estimator=DecisionTreeClassifier(), n_estimators=500,
max_samples=0.25,
bootstrap=True,
random_state=42)
oob_score =True,

Random Patches (JUST subsets of columns + bootstrap features)

OOB Score

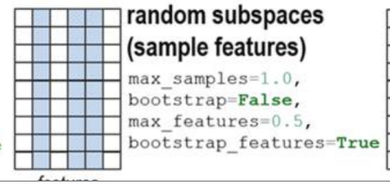
Regressor

Bagging compared to random subspaces and random patches.



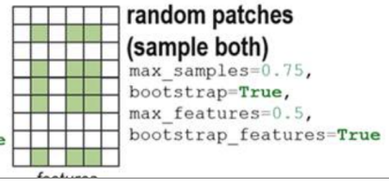
bagging (sample instances)

```
max_samples=0.75,
bootstrap=True,
max_features=1.0,
bootstrap_features=False
```



random subspaces (sample features)

```
max_samples=1.0,
bootstrap=False,
max_features=0.5,
bootstrap_features=True
```



random patches (sample both)

```
max_samples=0.75,
bootstrap=True,
max_features=0.5,
bootstrap_features=True
```

Voting

```
voting_clf_hard = VotingClassifier(estimators=[('lr', log_clf),
                                               ('svc', svc_clf),
                                               ('naive', naive_clf)],
                                   voting='hard')


voting_clf_soft = VotingClassifier(estimators=[('lr', log_clf),
                                                ('svc', svc_clf_soft),
                                                ('naive', naive_clf)],
                                   voting='soft' ,
                                   weights = [0.40, 0.40, 0.2])


# regrresor voting
from sklearn.ensemble import VotingRegressor
voting_regressor = VotingRegressor(estimators=[('xgb', search), ('lgbm', model), ('cb', modelCB)])
voting_regressor.fit(X_train, y_train)
voting_regressor.score(X_test, y_test)
```

Voting

Hard voting classifier (voting='hard') aggregate the predictions of each classifier and predict the class that gets the most votes

Soft voting classifier (voting='soft') predict the class with the highest class probability, averaged over all the individual classifiers.

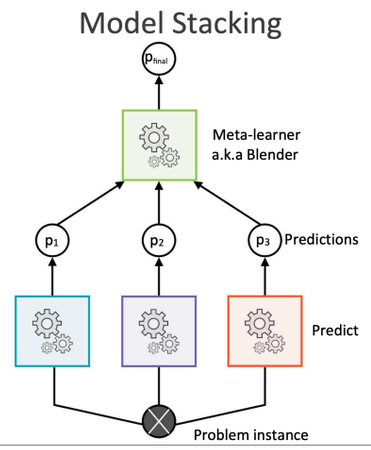
Stacking

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
estimators = [
    ('XGB', search),
    ('LGBM', clf),
    ('catb', clfCB)
    ]
stc= StackingClassifier(estimators=estimators, final_estimator=LogisticRegression())
stc.fit(X_train, y_train)
test_pred = stc.predict(X_test)
print(accuracy_score(y_test, test_pred))
```

```
from sklearn.ensemble import StackingRegressor
from sklearn.linear_model import LogisticRegression
estimators = [
    ('xgb', search), ('lgbm', model), ('cb', modelCB)
    ]
stacked_model = StackingRegressor(estimators=estimators, final_estimator=LogisticRegression())
stacked_model.fit(X_train, y_train)
stacked_model.score(X_test, y_test)
```

Stacking

```
from sklearn.ensemble import StackingRegressor
from sklearn.linear_model import LogisticRegression
estimators = [
    ('xgb', search), ('lgbm', model), ('cb', modelCB)
    ]
stacked_model = StackingRegressor(estimators=estimators, final_estimator=modelCB)
stacked_model.fit(X_train, y_train)
stacked_model.score(X_test, y_test)
```

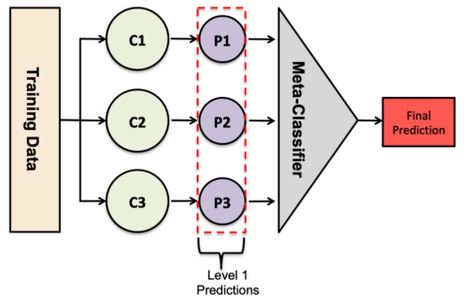


Stacking

Step 1 : In the first step, the individual classification models are trained based on the complete training set and their individual outputs are stored. These individual classification models are referred to as **Level One or Base Classifiers**.

Step 2 : In the second step, the predictions of individual classifiers (referred to as **Level One or Base Classifiers**) are used as new features to train a new classifier. This new classifier is called **Meta Classifier**. The meta-classifier can be any classifier of our choice.

The meta-classifier is fitted based on the outputs -- **meta-features** -- of the individual classification models in the ensemble. The meta-classifier can either be trained on the predicted class labels or probabilities from the ensemble. The figure below shows how three different classifiers get trained. Their predictions get stacked and are used as features to train the meta-classifier which makes the final prediction.



RANDOM FOREST EXTRA TREE:-CODE EXAMPLE

```
from sklearn.ensemble import ExtraTreesRegressor
extra_reg = ExtraTreesRegressor(n_estimators=600, max_leaf_nodes=12,
n_jobs=-1)
extra_reg.fit(x_train, y_train)
```

```
extra_clf = ExtraTreesClassifier(n_estimators = 600,
                                 max_leaf_nodes = 16,
                                 n_jobs = -1)

y_pred = extra_clf.predict(x_test)

accuracy_score(y_test, y_pred)
```

RANDOM FOREST EXTRA TREE:

Ensemble of decision trees

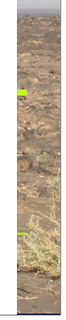
Usually assembled using bagging

During training

Random subset of data

Random subset of features at each

split point



XG BOOST:-extreme gradient boost

Built on top of gradient boost. He is faster because of his multi tasking (cv-that fixes itself)and better because of his learning rate.

Parameter Name	Parameter Type	Recommended Ranges
alpha	ContinuousParameterRanges	MinValue: 0, MaxValue: 1000
colsample_bylevel	ContinuousParameterRanges	MinValue: 0.1, MaxValue: 1
colsample_bynode	ContinuousParameterRanges	MinValue: 0.1, MaxValue: 1
colsample_bytree	ContinuousParameterRanges	MinValue: 0.5, MaxValue: 1
eta	ContinuousParameterRanges	MinValue: 0.1, MaxValue: 0.5
gamma	ContinuousParameterRanges	MinValue: 0, MaxValue: 5
lambda	ContinuousParameterRanges	MinValue: 0, MaxValue: 1000
max_delta_step	IntegerParameterRanges	[0, 10]
max_depth	IntegerParameterRanges	[0, 10]
min_child_weight	ContinuousParameterRanges	MinValue: 0, MaxValue: 120
num_round	IntegerParameterRanges	[1, 4000]
subsample	ContinuousParameterRanges	MinValue: 0.5, MaxValue: 1

XG BOOST:-extreme gradient boost

Params:

Parameters for XGBOOST

```
XGBClassifier(base_score=0.5, booster=None, colsample_bylevel=1,
              colsample_bynode=1, colsample_bytree=1, gamma=0, gpu_id=-1,
              importance_type='gain', interaction_constraints=None,
              learning_rate=0.300000012, max_delta_step=0, max_depth=6,
              min_child_weight=1, missing=nan, monotone_constraints=None,
              n_estimators=100, n_jobs=0, num_parallel_tree=1,
              objective='binary:logistic', random_state=0, reg_alpha=0,
              reg_lambda=1, scale_pos_weight=1, subsample=1, tree_method=None,
              validate_parameters=False, verbosity=None)
```

booster[default = gbtree]

- **booster** parameter helps us to choose which booster to use.
- It helps us to select the type of model to run at each iteration.
- It has 3 options - **gbtree**, **gblinear** or **dart**.
  - **gbtree** and **dart** - use tree-based models, while
  - **gblinear** uses linear models.

XG BOOST:-extreme gradient boost

Params:

nthread [default = maximum number of threads available if not set]

- This is number of parallel threads used to run XGBoost.
- This is used for parallel processing and number of cores in the system should be entered.
- If you wish to run on all cores, value should not be entered and algorithm will detect automatically

2.2 Booster Parameters ¶

Table of Contents

- We have 2 types of boosters - **tree booster** and **linear booster**.
- We will limit our discussion to **tree booster** because it always outperforms the **linear booster** and thus the later is rarely used.

eta [default=0.3, alias: learning_rate]

- It is analogous to learning rate in GBM.
- It is the step size shrinkage used in update to prevent overfitting.
- After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
- It makes the model more robust by shrinking the weights on each step.
- range : [0,1]
- Typical final values : 0.01-0.2.

2.2.2 gamma ¶

Table of Contents

- **gamma [default=0, alias: min_split_loss]**
  - A node is split only when the resulting split gives a positive reduction in the loss function.
  - Gamma specifies the minimum loss reduction required to make a split.
  - It makes the algorithm conservative. The values can vary depending on the loss function and should be tuned.
  - The larger gamma is, the more conservative the algorithm will be.
  - Range: [0,∞]

max_depth [default=6]

- The maximum depth of a tree, same as GBM.
- It is used to control over-fitting as higher depth will allow model to learn relations very specific to a particular sample.
- Increasing this value will make the model more complex and more likely to overfit.
- The value 0 is only accepted in lossguided growing policy when tree_method is set as hist and it indicates no limit on depth.
- We should be careful when setting large value of max_depth because XGBoost aggressively consumes memory when training a deep tree.
- range: [0,∞] (0 is only accepted in lossguided growing policy when tree_method is set as hist.
- Should be tuned using CV.
- Typical values: 3-10

XG BOOST:-extreme gradient boost

```
from sklearn.model_selection import GridSearchCV,RandomizedSearchCV
import xgboost as xgb
from sklearn.metrics import r2_score, mean_squared_error

param_search = {
 'max_depth':(5,6,7,8),
 'min_child_weight': (1,2,3,4,5),
 'learning_rate': (0.01, 0.03, 0.05,0.3,0.4),
 'subsample': (0.7, 0.8, 0.9, 1.0)
}
estimator = xgb.XGBRegressor()
search = RandomizedSearchCV(estimator, param_search, cv=6, n_iter=6, n_jobs=-1)


search.fit(X_train, y_train)


from sklearn.model_selection import GridSearchCV
import xgboost as xgb
param_search = {
 'max_depth':range(4,10),
 'min_child_weight': (1,3,5),
 'learning_rate': (0.01, 0.03, 0.001),
 'subsample': (0.4,0.5,0.6,0.7, 0.8, 0.9, 1.0),
 'n_estimators': (100,150,200,300),
}
estimator = xgb.XGBClassifier()
search = GridSearchCV(estimator, param_search, cv=10, n_jobs=-1)


search.fit(X_train, y_train)
print(search.score(X_test, y_test))
```

CATBOOSTCLASSIFIER:

CatBoost is an algorithm for **gradient boosting on decision trees.**

The goal of training is to select the *model*, depending on a set of *features* , that best solves the given problem (regression, classification, or multiclassification) for any input *object*. This model is found by using a *training dataset*, which is a set of objects with known features and label values. Accuracy is checked on the *validation dataset*, which has data in the same format as in the training dataset, but it is only used for evaluating the quality of training (it is not used for training).

CatBoost is based on gradient boosted decision trees. During training, a set of decision trees is built consecutively. Each successive tree is built with reduced loss compared to the previous trees.

The number of trees is controlled by the starting parameters. To prevent overfitting, use the overfitting detector. When it is triggered, trees stop being built.

- Preliminary calculation of splits.
- (*Optional*) Transforming categorical features to numerical features.
- (*Optional*) Transforming text features to numerical features.
- Choosing the tree structure. This stage is affected by the set Bootstrap options.
- Calculating values in leaves.

CATBOOST-params:

Parameter Name	Parameter Type	Recommended Ranges
learning_rate	ContinuousParameterRanges	MinValue: 0.001, MaxValue: 0.01
depth	IntegerParameterRanges	MinValue: 4, MaxValue: 10
l2_leaf_reg	IntegerParameterRanges	MinValue: 2, MaxValue: 10
random_strength	ContinuousParameterRanges	MinValue: 0, MaxValue: 10

3 MAINLY PARAMS THAT WORTH TO CHECK FOR BETTER ACCURACY:
Learning_rate
Iterations
random_strength

argument	description
iterations=500	The maximum number of trees that can be built when solving machine learning problems. Fewer may be used.
learning_rate=0.03	used for reducing the gradient step. It affects the overall time of training: the smaller the value, the more iterations are required for training.
depth=6	Depth of the tree. Can be any integer up to 32. Good values in the range 1 - 10.
l2_leaf_reg=3	try different values for the regularizer to find the best possible. Any positive values are allowed.
loss_function='Logloss'	For 2-class classification use 'LogLoss' or 'CrossEntropy'. For multiclass use 'MultiClass'.
border_count=32	The number of splits for numerical features. Allowed values are integers from 1 to 255 inclusively.
ctr_border_count=50	The number of splits for categorical features. Allowed values are integers from 1 to 255 inclusively.

CATBOOST:Examples

```
import catboost as cb
train_dataset = cb.Pool(X_train, y_train)
test_dataset = cb.Pool(X_test, y_test)
clf = cb.CatBoostClassifier(learning_rate=0.01,depth=10,leaf_estimation_method='Gradient',
boosting_type='Plain',iterations=300,random_strength=10)
clf.fit(train_dataset)
```

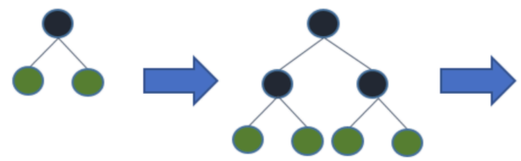
```
import catboost as cb
train_dataset = cb.Pool(X_train, y_train)
test_dataset = cb.Pool(X_test, y_test)
model =
cb.CatBoostRegressor(loss_function='RMSE',boosting_type='Plain',depth=8,learning_rate=0.03
,leaf_estimation_method='Gradient')
model.fit(train_dataset, eval_set=test_dataset, use_best_model=True)-> u can add plot=True
```

https://catboost.ai/en/docs/references/training-parameters/common#loss_function

LGBM- Light gradient boostibg

LightGBM is a gradient boosting framework that uses tree based learning algorithms. It is designed to be distributed and efficient with the following advantages:

- Faster training speed and higher efficiency.
- Lower memory usage.
- Better accuracy.
- Support of parallel, distributed, and GPU learning.
- Capable of handling large-scale data.



Level-wise tree growth

https://lightgbm.readthedocs.io/en/v3.3.2/

LGBM- Params:

- In this section, I will discuss some tips to improve LightGBM model efficiency.
- Following set of practices can be used to improve your model efficiency.
  - **1 num_leaves : This is the main parameter to control the complexity of the tree model. Ideally, the value of num_leaves should be less than or equal to 2^(max_depth). Value more than this will result in overfitting.**
  - **2 min_data_in_leaf : Setting it to a large value can avoid growing too deep a tree, but may cause under-fitting. In practice, setting it to hundreds or thousands is enough for a large dataset.**
  - **3 max_depth : We also can use max_depth to limit the tree depth explicitly.**

For Faster Speed

- Use bagging by setting bagging_fraction and bagging_freq.
- Use feature sub-sampling by setting feature_fraction.
- Use small max_bin.
- Use save_binary to speed up data loading in future learning.

For better accuracy

Learning rate-most significant.

- Use large max_bin (may be slower).
- Use small learning_rate with large num_iterations-significant
- Use large num_leaves(may cause over-fitting)
- Use bigger training data-significant
- Try dart
- Try to use categorical feature directly.

To deal with over-fitting¶

- Use small max_bin
- Use small num_leaves
- Use min_data_in_leaf and min_sum_hessian_in_leaf
- Use bagging by set bagging_fraction and bagging_freq
- Use feature sub-sampling by set feature_fraction
- Use bigger training data
- Try lambda_l1, lambda_l2 and min_gain_to_split to regularization
- Try max_depth to avoid growing deep tree

https://lightgbm.readthedocs.io/en/v3.3.2/

https://www.kaggle.com/code/prashant111/lightgbm-classifier-in-python

LGBM- Params:

Parameter Name	Parameter Type	Recommended Ranges
learning_rate	ContinuousParameterRanges	MinValue: 0.001, MaxValue: 0.01
num_leaves	IntegerParameterRanges	MinValue: 10, MaxValue: 100
feature_fraction	ContinuousParameterRanges	MinValue: 0.1, MaxValue: 1.0
bagging_fraction	ContinuousParameterRanges	MinValue: 0.1, MaxValue: 1.0
bagging_freq	IntegerParameterRanges	MinValue: 0, MaxValue: 10
max_depth	IntegerParameterRanges	MinValue: 15, MaxValue: 100
min_data_in_leaf	IntegerParameterRanges	MinValue: 10, MaxValue: 200

https://lightgbm.readthedocs.io/en/v3.3.2/

https://www.kaggle.com/code/prashant111/lightgbm-classifier-in-python

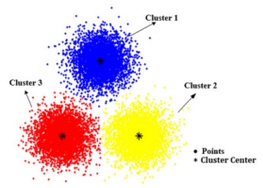
LGBM- Params:

```
params = {
    'task': 'train', # train or predict
    'boosting': 'gbdt',
    'objective': 'regression',# when the target is a regression problem
    'num_leaves': 10,# controls the size of the trees
    'learnnig_rage': 0.05,# for better accuracey, the model learn from him self
    'metric': {'l2','l1'},# the metric to evaluate the model
    'silent':True,# varbose off
    'min_data_in_leaf': 40},# avoids overfitting

# laoding data
lgb_train = lgb.Dataset(X_train, y_train)
lgb_eval = lgb.Dataset(X_test, y_test, reference=lgb_train)

model = lgb.train(params,
                  train_set=lgb_train,
                  valid_sets=lgb_eval,
                  early_stopping_rounds=30) # if the model doesn't improve it stops..


import lightgbm as lgb
clf = lgb.LGBMClassifier()
clf.fit(X_train, y_train)
```

https://lightgbm.readthedocs.io/en/v3.3.2/

https://www.kaggle.com/code/prashant111/lightgbm-classifier-in-python

K-means-clusterin



Identify patterns in data items e.g. K-means clustering

```
Initialise K centroids


Repeat:
  For each data point:
    Assign to "nearest" cluster


  For each centroid:
    Update coordinates

Have centroids converged?
   Yes: Stop, we're done
    No: Keep iterating
```

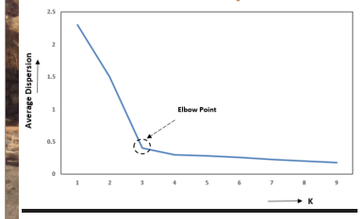
- Pick an initial solution (algorithms exist to pick well)
- Iterate until convergence
  - Update assignments of points to clusters
  - Update coordinates of reference vectors
- Keep iterating until we converge

K-means-clusterin

1. Choose the number of clusters K.
2. Select at random K points, the centroids(not necessarily from your dataset).
3. Assign each data point to the closest centroid → that forms K clusters.
4. Compute the mean of data points within a cluster and place the new centroid of each cluster.
5. Reassign each data point to the new closest centroid. If any reassignment . took place, go to step 4, otherwise, the model is ready.

How to choose the value of K?

There are different techniques available to find the optimal value of K. The most common technique is the **elbow method.
The elbow method is used to determine the optimal number of clusters in K-means clustering. The elbow method plots the value of the Within cluster sum of square of the data points(WCSS) produced by different values of K. The below diagram shows how the elbow method works:-



K-means-clusterin

- Hyperparameters
  - Number of clusters
  - Initial values of centroids

Homogeneity	Completeness	V-measure
Adjusted Rand Index (ARI)	Adjusted Mutual Info	Silhouette

K-means-clusterin-CODE

```
k_means3 = KMeans(n_clusters=3,max_iter = 400, n_init = 10, random_state=7)

k_means3.fit(X)

labels = k_means3.labels_
# inertia
print("Inertia: ",k_means3.inertia_)

# silhouette score
silhouette_avg3 = silhouette_score(X, (k_means3.labels_))
print("The silhouette score is ", silhouette_avg3)


from sklearn.metrics import silhouette_score

# Silhouette analysis
range_n_clusters = [2, 3, 4, 5, 6]

for num_clusters in range_n_clusters:

    # intialise kmeans
    kmeans = KMeans(n_clusters=num_clusters, max_iter=250)
    kmeans.fit(X)

    cluster_labels = kmeans.labels_

    # silhouette score
    silhouette_avg = silhouette_score(X, cluster_labels)#<-----------------------
    print("For n_clusters={0}, the silhouette score is {1}".format(num_clusters, silhouette_avg))


cs = []
for i in range(1, 11):
    kmeans = KMeans(n_clusters = i, init = 'k-means++', max_iter = 300, n_init = 10, random_state = 7)

    kmeans.fit(X)

    cs.append(kmeans.inertia_)

# plot the
plt.plot(range(1, 11), cs)
plt.title('The Elbow Method')
plt.xlabel('Number of clusters')
plt.ylabel('Inertia')
plt.show()
```

Elbow graph

Confusion Matrix

	Actual = Yes	Actual = No
Predicted = Yes	TP	FP
Predicted = No	FN	TN

Confusion matrix

True positive
זיהוי נכון של דוגמא חיובית

False Positive = False Alarm
זיהוי של דוגמא שלילית כחיובית

True negative
זיהוי נכון של דוגמא שלילית

False Negative = Miss Detection
זיהוי של דוגמא חיובית כשלילית

Confusion Matrix-code example

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

```
from sklearn.metrics import classification_report, confusion_matrix, plot_confusion_matrix
plot_confusion_matrix(clf, X_test, y_test,values_format="d",cmap='Blues');
```

ROC CURVE

ROC curve

An **ROC curve** (**receiver operating characteristic curve**) is a graph showing the performance of a classification model at all classification thresholds. This curve plots two parameters:

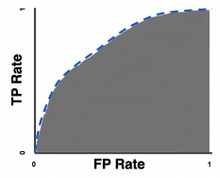
- True Positive Rate
- False Positive Rate

True Positive Rate (**TPR**) is a synonym for recall and is therefore defined as follows:

$$TPR = \frac{TP}{TP + FN}$$

False Positive Rate (**FPR**) is defined as follows:

$$FPR = \frac{FP}{FP + TN}$$



COOL stuff:

counter of vectors:

```
# creating a count matrix
cv = CountVectorizer()
```

ROC CORVE

```
#roc curve
from sklearn.metrics import roc_curve, auc
false_positive_rate, true_positive_rate, thresholds = roc_curve(y_test, y_pred)
roc_auc = auc(false_positive_rate, true_positive_rate)
plt.figure(figsize=[10,10])
plt.title('Receiver Operating Characteristic')
plt.plot(false_positive_rate, true_positive_rate, 'b',
label='AUC = %0.2f'% roc_auc)
plt.legend(loc='lower right')
plt.plot([0,1],[0,1],'r--')
plt.xlim([0,1])
plt.ylim([0,1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

Label encoder:

```
from sklearn.preprocessing import
LabelEncoder
labelencoder_y = LabelEncoder()
y = labelencoder_y.fit_transform(y)
# [1,1,1,1,1,1,1,0,0,0,0,0,0,0,0]
```

High corr

```
#find highly correlated features
def correlation(dataset, threshold):
    col_corr = set()
    corr_matrix = dataset.corr()
    for i in
range(len(corr_matrix.columns)):
        for j in range(i):
            if abs(corr_matrix.iloc[i, j])
>= threshold and (corr_matrix.columns[j]
not in col_corr):
                colname =
corr_matrix.columns[i]
                col_corr.add(colname)
    return col_corr
```