How did it do that ?

STEP 1: Choose the number K of neighbors

STEP 2: Take the K nearest neighbors of the new data point, according to the Euclidean distance

STEP 3: Among these K neighbors, count the number of data points in each category

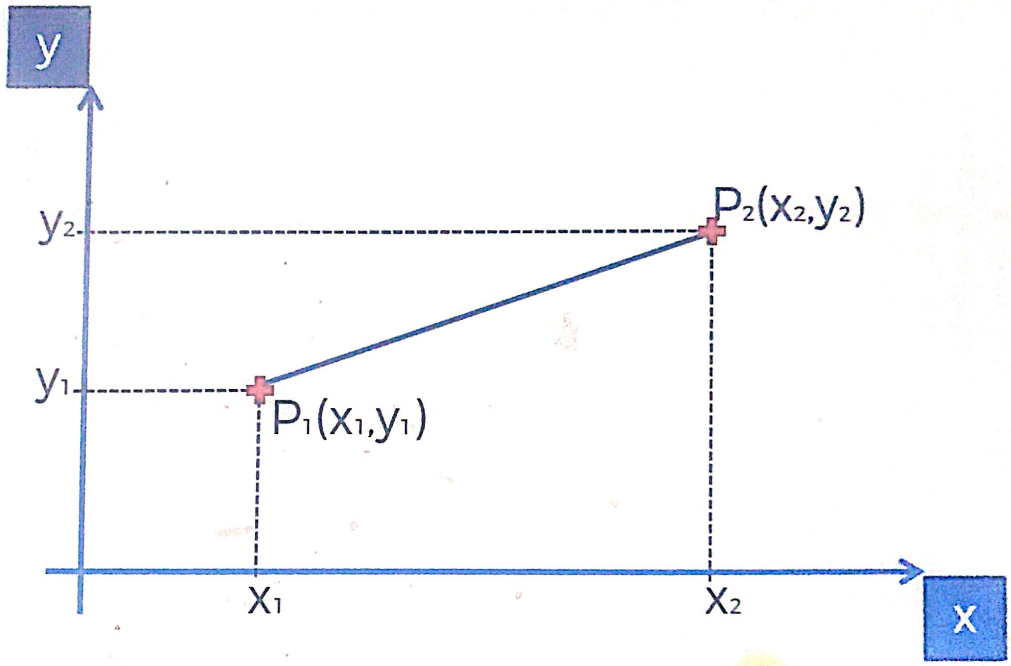
STEP 4: Assign the new data point to the category where you counted the most neighbors

Your Model is Ready

IN algorithm

STEP 1: Choose the number K of neighbors: K = 5

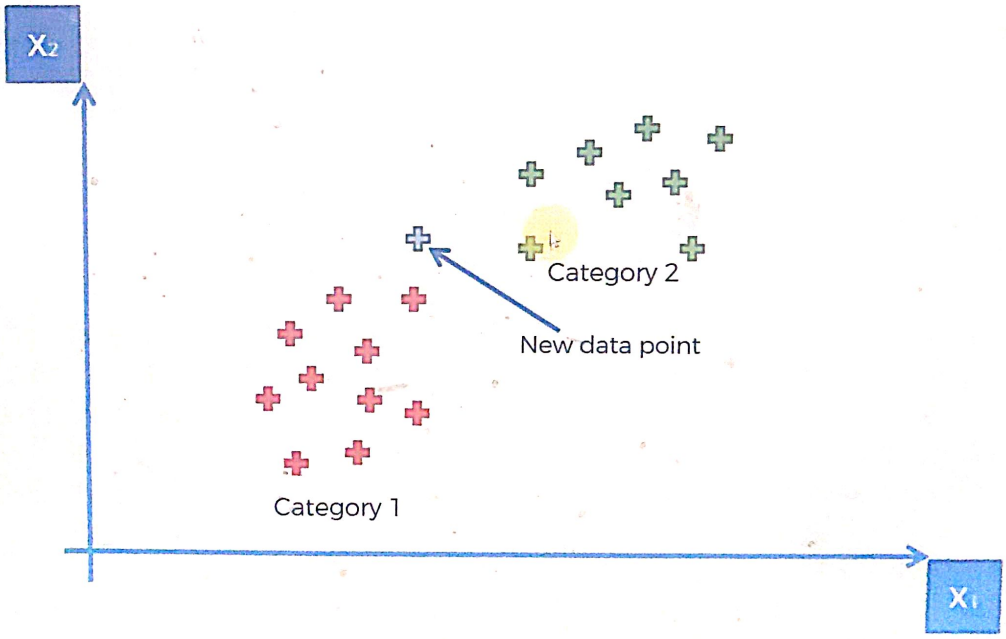
Euclidean Distance



Euclidean Distance between P_1 and $P_2 = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$

K-NN algorithm

STEP 2: Take the K = 5 nearest neighbors of the new data point, according to the Euclidean distance



K-NN algorithm

STEP 3: Among these K neighbors, count the number of data points in each category

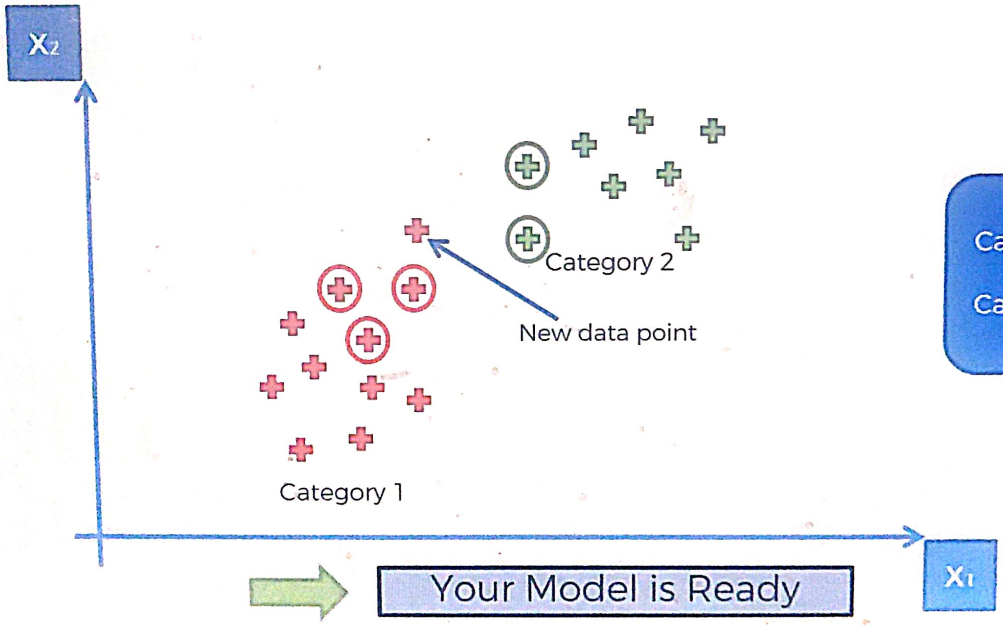
K-NN algorithm

STEP 4: Assign the new data point to the category where you counted the most neighbors

assign the new data point to the category, where u count the most neighbours.
that means we need to assign the red category here

K-NN algorithm

STEP 4: Assign the new data point to the category where you counted the most neighbors



Category 1: 3 neighbors

Category 2: 2 neighbors

Social_Network_Ads.csv

Age	EstimatedSalary	Purchased
19	19000	0
35	20000	0
26	43000	0
27	57000	0
19	76000	0
27	58000	0
27	84000	0
32	150000	1
25	33000	0
35	65000	0
26	80000	0
26	52000	0
20	86000	0
32	18000	0
18	82000	0
29	80000	0
47	25000	1
45	26000	1
46	28000	1
48	29000	1
45	22000	1
47	49000	1
48	41000	1
45	22000	1
46	23000	1
47	20000	1
49	28000	1
47	30000	1
29	43000	0
31	18000	0
31	74000	0
27	137000	1

Open in playground

Training the K-NN model on the Training set

```
[ ]  1 from sklearn.neighbors import KNeighborsClassifier
     2 classifier = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski', p = 2)
     3 classifier.fit(X_train, y_train)
```

```
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='minkowski',
                     metric_params=None, n_jobs=None, n_neighbors=5, p=2,
                     weights='uniform')
```

API Reference

This is the class and function reference of scikit-learn. Please refer to the full user guide for further details, as the class and function raw specifications may not be enough to give full guidelines on their uses. For reference on concepts repeated across the API, see Glossary of Common Terms and API Elements.

sklearn.base: Base classes and utility functions

Base classes for all estimators.

Used for VotingClassifier

Base classes

`base.BaseEstimator`	Base class for all estimators in scikit-learn
`base.BiclusterMixin`	Mixin class for all bicluster estimators in scikit-learn
`base.ClassifierMixin`	Mixin class for all classifiers in scikit-learn.
`base.ClusterMixin`	Mixin class for all cluster estimators in scikit-learn.
`base.DensityMixin`	Mixin class for all density estimators in scikit-learn.
`base.RegressorMixin`	Mixin class for all regression estimators in scikit-learn.
`base.TransformerMixin`	Mixin class for all transformers in scikit-learn.

Functions

`base.clone`(estimator[, safe])	Constructs a new estimator with the same parameters.
`base.is_classifier`(estimator)	Return True if the given estimator is (probably) a classifier.
`base.is_regressor`(estimator)	Return True if the given estimator is (probably) a regressor.
`config_context`(**new_config)	Context manager for global scikit-learn configuration
`get_config`()	Retrieve current values for configuration set by `set_config`
`set_config`([assume_finite, working_memory, ...])	Set global scikit-learn configuration
`show_versions`()	Print useful debugging information

sklearn.calibration: Probability Calibration

Calibration of predicted probabilities.

User guide: See the Probability calibration section for further details.

`calibration.CalibratedClassifierCV`([...])	Probability calibration with isotonic regression or sigmoid.
`calibration.calibration_curve`(y_true, y_prob)	Compute true and predicted probabilities for a calibration curve.

sklearn.cluster: Clustering

The `sklearn.cluster` module gathers popular unsupervised clustering algorithms.

User guide: See the Clustering and Biclustering sections for further details.

Classes

`cluster.AffinityPropagation`([damping, ...])	Perform Affinity Propagation Clustering of data.
`cluster.AgglomerativeClustering`([...])	Agglomerative Clustering
`cluster.Birch`([threshold, branching_factor, ...])	Implements the Birch clustering algorithm.
`cluster.DBSCAN`([eps, min_samples, metric, ...])	Perform DBSCAN clustering from vector array or distance matrix.
`cluster.FeatureAgglomeration`([n_clusters, ...])	Agglomerate features.
`cluster.KMeans`([n_clusters, init, n_init, ...])	K-Means clustering
`cluster.MiniBatchKMeans`([n_clusters, init, ...])	Mini-Batch K-Means clustering
`cluster.MeanShift`([bandwidth, seeds, ...])	Mean shift clustering using a flat kernel.

The `sklearn.neighbors` module implements the k-nearest neighbors algorithm.

User guide: See the Nearest Neighbors section for further details.

`neighbors.BallTree`	BallTree for fast generalized N-point problems
`neighbors.DistanceMetric`	DistanceMetric class
`neighbors.KDTree`	KDTree for fast generalized N-point problems
`neighbors.KernelDensity([bandwidth, ...])`	Kernel Density Estimation.
`neighbors.KNeighborsClassifier([...])`	Classifier implementing the k-nearest neighbors vote.
`neighbors.KNeighborsRegressor([n_neighbors, ...])`	Regression based on k-nearest neighbors.
`neighbors.KNeighborsTransformer([mode, ...])`	Transform X into a (weighted) graph of k nearest neighbors
`neighbors.LocalOutlierFactor([n_neighbors, ...])`	Unsupervised Outlier Detection using Local Outlier Factor (LOF)
`neighbors.RadiusNeighborsClassifier([...])`	Classifier implementing a vote among neighbors within a given radius
`neighbors.RadiusNeighborsRegressor([radius, ...])`	Regression based on neighbors within a fixed radius.
`neighbors.RadiusNeighborsTransformer([mode, ...])`	Transform X into a (weighted) graph of neighbors nearer than a radius
`neighbors.NearestCentroid([metric, ...])`	Nearest centroid classifier.
`neighbors.NearestNeighbors([n_neighbors, ...])`	Unsupervised learner for implementing neighbor searches.
`neighbors.NeighborhoodComponentsAnalysis([...])`	Neighborhood Components Analysis

`neighbors.kneighbors_graph(X, n_neighbors[, ...])`	Computes the (weighted) graph of k-Neighbors for points in X
`neighbors.radius_neighbors_graph(X, radius)`	Computes the (weighted) graph of Neighbors for points in X

`sklearn.neural_network`: Neural network models

The `sklearn.neural_network` module includes models based on neural networks.

User guide: See the Neural network models (supervised) and Neural network models (unsupervised) sections for further details.

`neural_network.BernoulliRBM([n_components, ...])`	Bernoulli Restricted Boltzmann Machine (RBM).
`neural_network.MLPClassifier([...])`	Multi-layer Perceptron classifier.
`neural_network.MLPRegressor([...])`	Multi-layer Perceptron regressor.

`sklearn.pipeline`: Pipeline

The `sklearn.pipeline` module implements utilities to build a composite estimator, as a chain of transforms and estimators.

`pipeline.FeatureUnion(transformer_list[, ...])`	Concatenates results of multiple transformer objects.
`pipeline.Pipeline(steps[, memory, verbose])`	Pipeline of transforms with a final estimator.
`pipeline.make_pipeline(*steps, **kwargs)`	Construct a Pipeline from the given estimators.
`pipeline.make_union(*transformers, **kwargs)`	Construct a FeatureUnion from the given transformers.

`sklearn.preprocessing`: Preprocessing and Normalization

The `sklearn.neighbors` module implements the k-nearest neighbors algorithm.

User guide: See the Nearest Neighbors section for further details.

All these are classes inside neighbours module of Scikit learn library

`neighbors.BallTree`	BallTree for fast generalized N-point problems
`neighbors.DistanceMetric`	DistanceMetric class
`neighbors.KDTree`	KDTree for fast generalized N-point problems
`neighbors.KernelDensity`([bandwidth, ...])	Kernel Density Estimation.
`neighbors.KNeighborsClassifier`([n_neighbors, ...])	Classifier implementing the k-nearest neighbors vote
`neighbors.KNeighborsRegressor`([n_neighbors, ...])	Regression based on k-nearest neighbors.
`neighbors.KNeighborsTransformer`([mode, ...])	Transform X into a (weighted) graph of k nearest neig
`neighbors.LocalOutlierFactor`([n_neighbors, ...])	Unsupervised Outlier Detection using Local Outlier Fa
`neighbors.RadiusNeighborsClassifier`([...])	Classifier implementing a vote among neighbors withi
`neighbors.RadiusNeighborsRegressor`([radius, ...])	Regression based on neighbors within a fixed radius.
`neighbors.RadiusNeighborsTransformer`([mode, ...])	Transform X into a (weighted) graph of neighbors nea
`neighbors.NearestCentroid`([metric, ...])	Nearest centroid classifier.
`neighbors.NearestNeighbors`([n_neighbors, ...])	Unsupervised learner for implementing neighbor searc
`neighbors.NeighborhoodComponentsAnalysis`([...])	Neighborhood Components Analysis

`neighbors.kneighbors_graph`(X, n_neighbors[, ...])	Computes the (weighted) graph of k-Neighbors for points
`neighbors.radius_neighbors_graph`(X, radius)	Computes the (weighted) graph of Neighbors for points in

sklearn.neighbors.KNeighborsClassifier

class sklearn.neighbors.KNeighborsClassifier(n_neighbors=5, weights='uniform', algorithm='auto', leaf_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None, **kwargs) [source]

Classifier implementing the k-nearest neighbors vote.

Read more in the User Guide.

Parameters:

n_neighbors : *int, optional (default = 5)*
Number of neighbors to use by default for `kneighbors` queries.

weights : *str or callable, optional (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.

algorithm : *{'auto', 'ball_tree', 'kd_tree', 'brute'}, optional*
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.

Note: fitting on sparse input will override the setting of this parameter, using brute force.

leaf_size : *int, optional (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 : *integer, optional (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 : *string or callable, default 'minkowski'*
the distance metric to use for the tree. The default metric is minkowski, and with p=2 is equivalent to the standard Euclidean metric. See the documentation of the DistanceMetric class for a list of available metrics. If metric is "precomputed", X is assumed to be a distance matrix and must be square during fit. X may be a Glossary, in which case only "nonzero" elements may be considered neighbors.

metric_params : *dict, optional (default = None)*
Additional keyword arguments for the metric function.

n_jobs : *int or None, optional (default=None)*
The number of parallel jobs to run for neighbors search. None means 1 unless in a joblib.parallel_backend context. -1 means using all processors. See Glossary for more details. Doesn't affect fit method.

Attributes:

classes_ : *array of shape (n_classes,)*
Class labels known to the classifier

effective_metric_ : *string or callable*
The distance metric used. It will be same as the metric parameter or a synonym of it, e.g. 'euclidean' if the metric parameter set to 'minkowski' and p parameter set to 2.

+ Code + Text

Training the K-NN model on the Training set

```
1 from sklearn.neighbors import KNeighborsClassifier
2 classifier = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski', p = 2)
```

Predicting a new result

k_neighbours=5 works best
metric kowasky is for Euclidean distance

```
1 print(classifier.predict(sc.transform([[30,87000]])))
```

```
[0]
```

Predicting the Test set results

```
1 y_pred = classifier.predict(X_test)
2 print(np.concatenate((y_pred.reshape(len(y_pred),1), y_test.reshape(len(y_test),1)),1))
```

```
[[0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [0 0]
 [1 1]
 [0 0]
 [1 0]
 [0 0]
```

Training the K-NN model on the Training set

```
from sklearn.neighbors import KNeighborsClassifier
classifier = KNeighborsClassifier(n_neighbors = 5, metric = 'minkowski', p = 2)
classifier.fit(X_train, y_train)
```

Visualising the Training set results

```
from matplotlib.colors import ListedColormap
X_set, y_set = sc.inverse_transform(X_train), y_train
X1, X2 = np.meshgrid(np.arange(start = X_set[:, 0].min() - 10, stop = X_set[:, 0].max() + 10, step = 1),
                     np.arange(start = X_set[:, 1].min() - 1000, stop = X_set[:, 1].max() + 1000, step = 1))
plt.contourf(X1, X2, classifier.predict(sc.transform(np.array([X1.ravel(), X2.ravel()]).T)).reshape(X1.shape),
             alpha = 0.75, cmap = ListedColormap(('red', 'green')))
plt.xlim(X1.min(), X1.max())
plt.ylim(X2.min(), X2.max())
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('K-NN (Training set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

Rest everything remains the same!!

```
for i, j in enumerate(np.unique(y_set)):
    plt.scatter(X_set[y_set == j, 0], X_set[y_set == j, 1], c = ListedColormap(('red', 'green'))(i), label = j)
plt.title('K-NN (Test set)')
plt.xlabel('Age')
plt.ylabel('Estimated Salary')
plt.legend()
plt.show()
```

```
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have pr
'c' argument looks like a single numeric RGB or RGBA sequence, which should be avoided as value-mapping will have pr
```