In [367…
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from imblearn.under_sampling import RandomUnderSampler
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, recall_score , precision_score, f1_scor
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import learning_curve
```

In [368…
```
magic_data = pd.read_csv('magic04.data', header=None)
magic_data.head()
```

Out[368…

	0	1	2	3	4	5	6	7	8	
0	28.7967	16.0021	2.6449	0.3918	0.1982	27.7004	22.0110	-8.2027	40.0920	81.88
1	31.6036	11.7235	2.5185	0.5303	0.3773	26.2722	23.8238	-9.9574	6.3609	205.26
2	162.0520	136.0310	4.0612	0.0374	0.0187	116.7410	-64.8580	-45.2160	76.9600	256.78
3	23.8172	9.5728	2.3385	0.6147	0.3922	27.2107	-6.4633	-7.1513	10.4490	116.73
4	75.1362	30.9205	3.1611	0.3168	0.1832	-5.5277	28.5525	21.8393	4.6480	356.46

In [369…
```
column_names = [
    'fLength', 'fWidth', 'fSize', 'fConc', 'fConc1',
    'fAsym', 'fM3Long', 'fM3Trans', 'fAlpha', 'fDist', 'class'
]
magic_data.columns = column_names
magic_data.head()
```

Out[369…

	fLength	fWidth	fSize	fConc	fConc1	fAsym	fM3Long	fM3Trans	fAlpha	
0	28.7967	16.0021	2.6449	0.3918	0.1982	27.7004	22.0110	-8.2027	40.0920	81.
1	31.6036	11.7235	2.5185	0.5303	0.3773	26.2722	23.8238	-9.9574	6.3609	205.
2	162.0520	136.0310	4.0612	0.0374	0.0187	116.7410	-64.8580	-45.2160	76.9600	256.
3	23.8172	9.5728	2.3385	0.6147	0.3922	27.2107	-6.4633	-7.1513	10.4490	116.
4	75.1362	30.9205	3.1611	0.3168	0.1832	-5.5277	28.5525	21.8393	4.6480	356.

In [370…
```
X = magic_data.drop('class', axis=1)
y = magic_data['class']
```

In [371…
```
X_g = X[y == 'g']
y_g = y[y == 'g']
```

```
X_h = X[y == 'h']
y_h = y[y == 'h']
```

In [372…

```
n_samples_h = len(X_h)
```

In [373…

```
rus = RandomUnderSampler(sampling_strategy={
    'g': n_samples_h,
    'h': n_samples_h
}, random_state=42)
```

In [374…

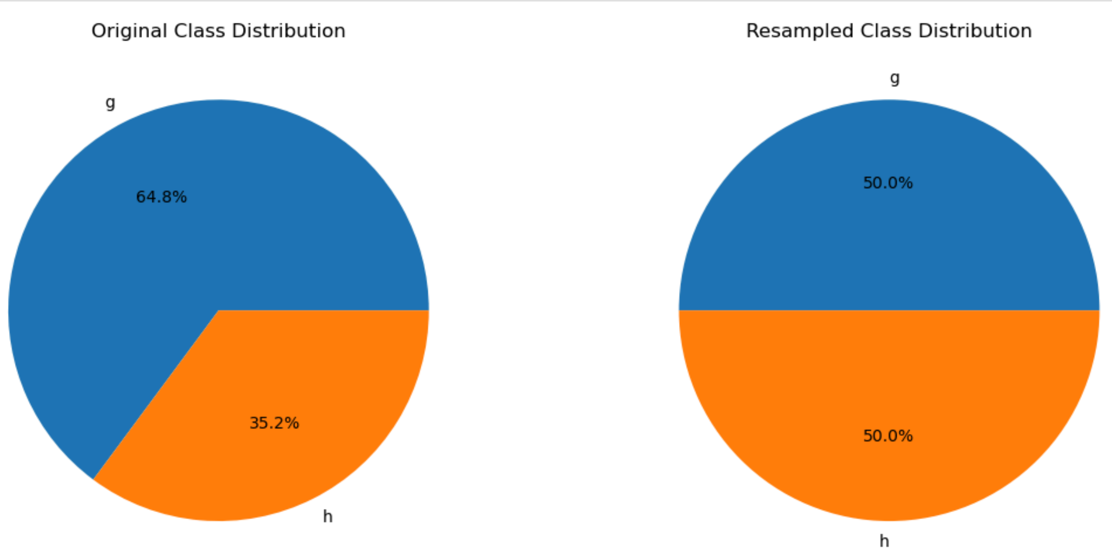
```
X_resampled, y_resampled = rus.fit_resample(X, y)
print(f"Original dataset shape: {X.shape}")
print(f"Resampled dataset shape: {X_resampled.shape}")
print(f"Original class distribution: {y.value_counts()}")
print(f"Resampled class distribution: {y_resampled.value_counts()}")
```

```
Original dataset shape: (19020, 10)
Resampled dataset shape: (13376, 10)
Original class distribution: class
g    12332
h     6688
Name: count, dtype: int64
Resampled class distribution: class
g    6688
h    6688
Name: count, dtype: int64
```

In [375…

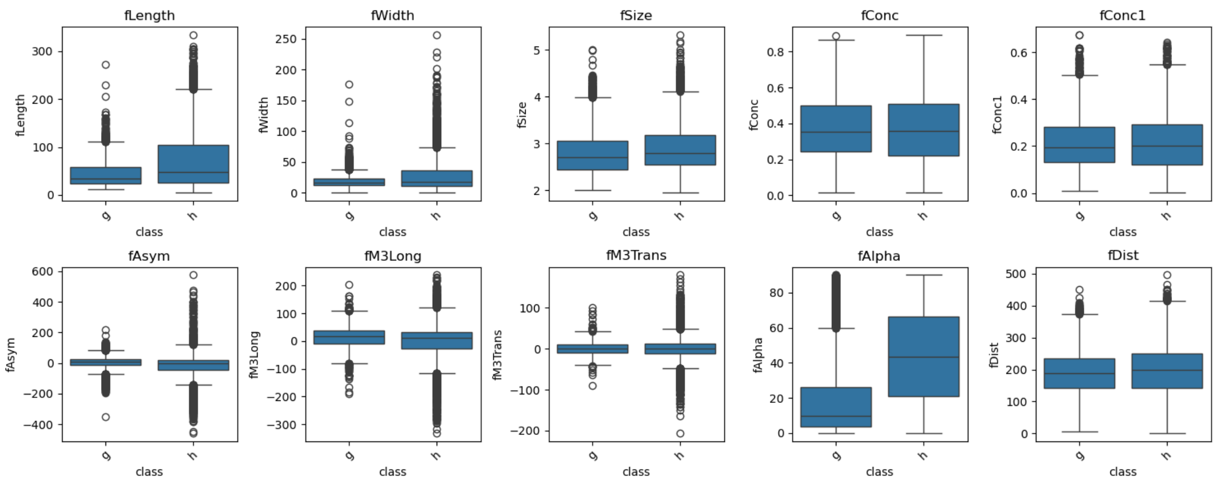
```
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.pie(y.value_counts(), labels=y.value_counts().index, autopct='%1.1f%%')
plt.title('Original Class Distribution')
plt.subplot(1, 2, 2)
plt.pie(y_resampled.value_counts(), labels=y_resampled.value_counts().index, autopc
plt.title('Resampled Class Distribution')

plt.tight_layout()
plt.show()
```



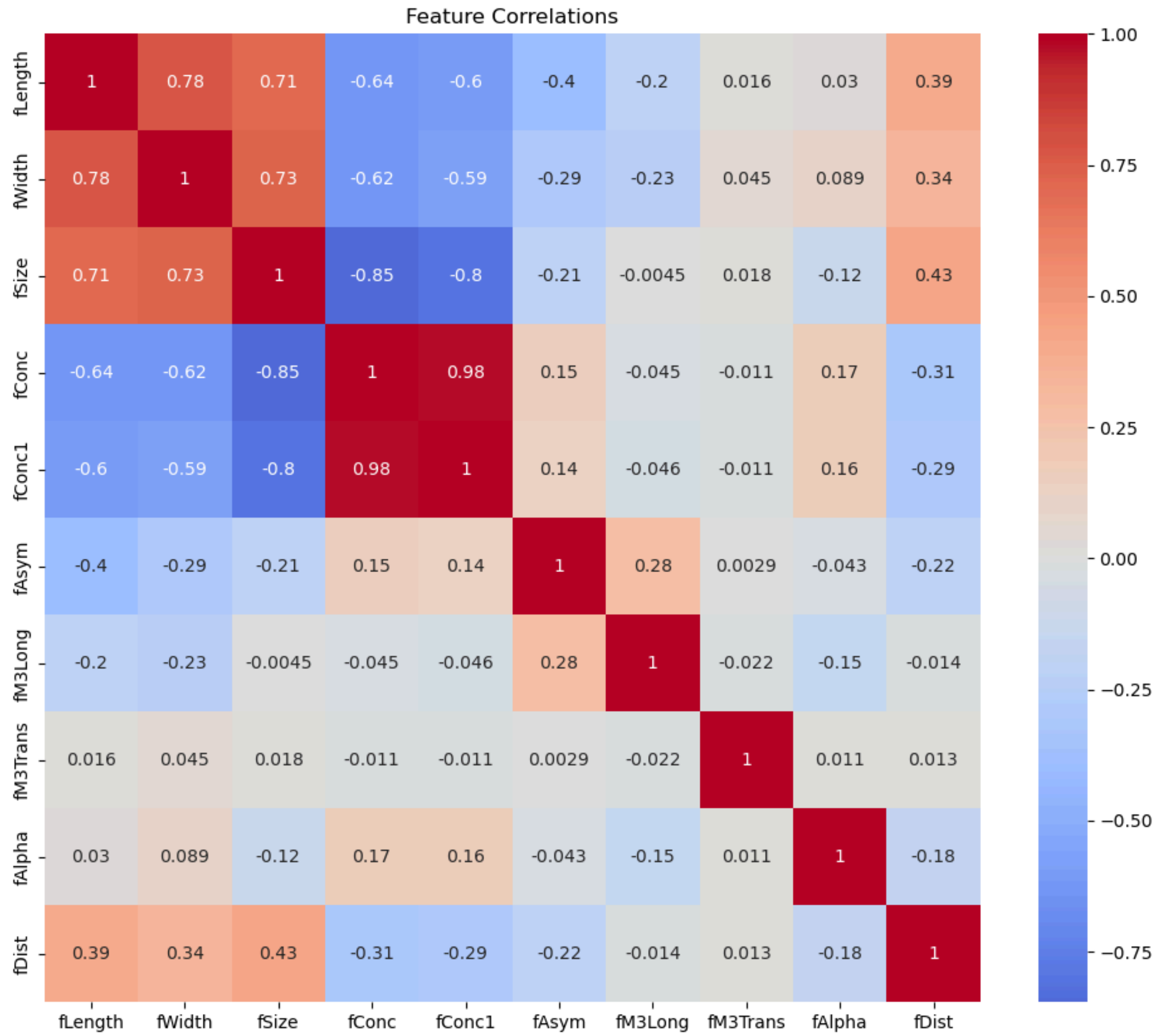
In [376…

```
plt.figure(figsize=(15, 6))
for i, feature in enumerate(X_resampled.columns):
    plt.subplot(2, 5, i+1)
    sns.boxplot(x=y_resampled, y=X_resampled[feature])
    plt.title(feature)
    plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



In [377…

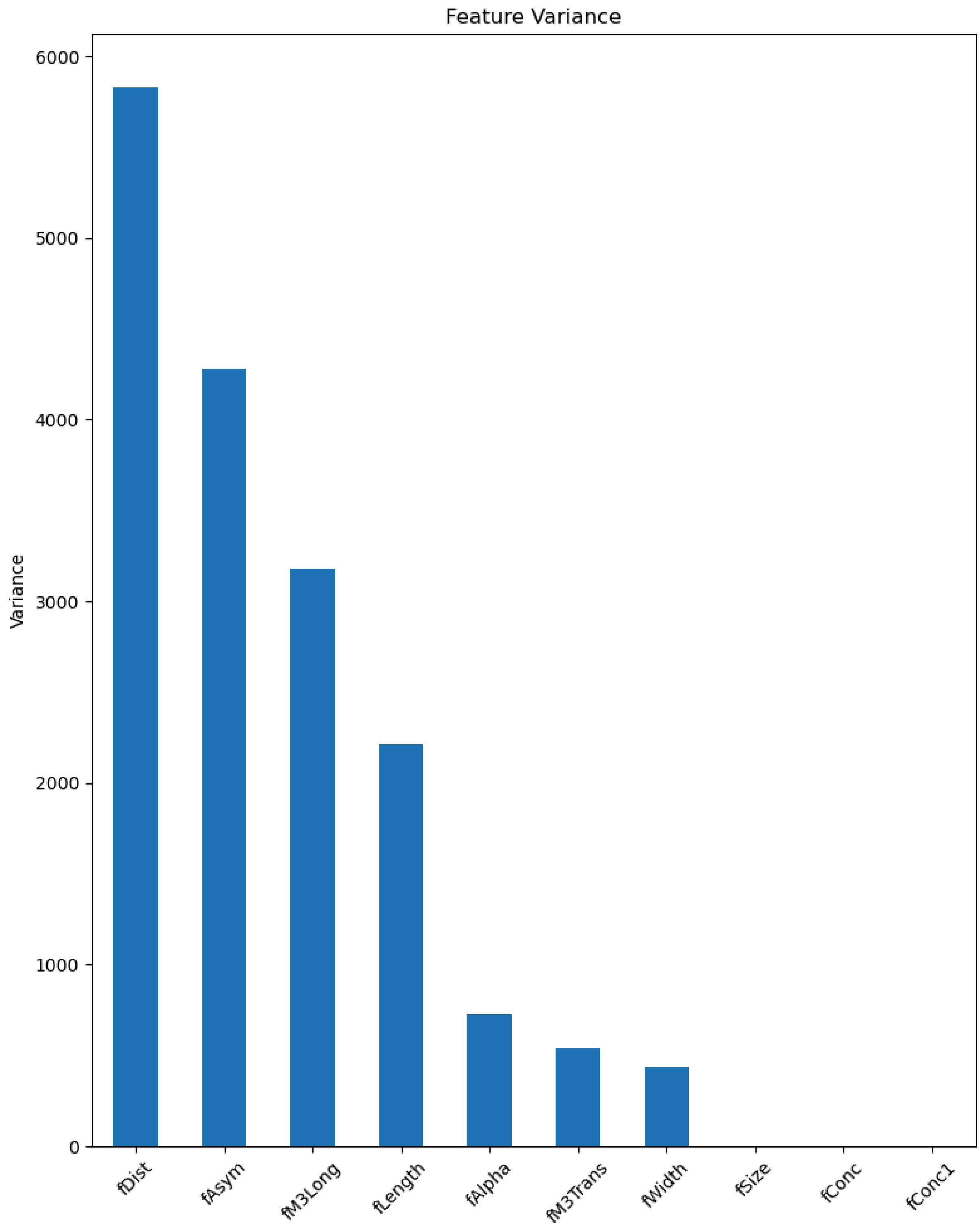
```
plt.figure(figsize=(12, 10))
correlation_matrix = X_resampled.corr()
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', center=0)
plt.title('Feature Correlations')
plt.show()
```



In [378…

```
feature_variance = X_resampled.var()

plt.figure(figsize=(8, 10))
feature_variance.sort_values(ascending=False).plot(kind='bar')
plt.title('Feature Variance')
plt.ylabel('Variance')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



In [379…

```
X_resampled_g = X_resampled[y_resampled == 'g']
y_resampled_g = y_resampled[y_resampled == 'g']

X_resampled_h = X_resampled[y_resampled == 'h']
y_resampled_h = y_resampled[y_resampled == 'h']
```

In [380…

```
X_train_val_g, X_test_g, y_train_val_g, y_test_g = train_test_split(X_resampled_g,
X_train_g, X_val_g, y_train_g, y_val_g = train_test_split(X_train_val_g, y_train_va
```

In [381…
```
X_train_val_h, X_test_h, y_train_val_h, y_test_h = train_test_split(X_resampled_h,
X_train_h, X_val_h, y_train_h, y_val_h = train_test_split(X_train_val_h, y_train_va
```

In [382…
```
X_train_scaled = np.concatenate((X_train_g, X_train_h), axis=0)
X_val_scaled = np.concatenate((X_val_g, X_val_h), axis=0)
X_test_scaled = np.concatenate((X_test_g, X_test_h), axis=0)

y_train = np.concatenate((y_train_g, y_train_h), axis=0)
y_val = np.concatenate((y_val_g, y_val_h), axis=0)
y_test = np.concatenate((y_test_g, y_test_h), axis=0)
```

In [383…
```
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_scaled)
X_val_scaled = scaler.transform(X_val_scaled)
X_test_scaled = scaler.transform(X_test_scaled)
```

In [384…
```
param_grid = {
    'n_neighbors': [1, 3, 5, 7, 9, 11, 13, 15, 17],
    'metric': ['euclidean', 'manhattan'],
}

knn = KNeighborsClassifier()
grid_search = GridSearchCV(knn,param_grid=param_grid,cv=5)
grid_search.fit(X_train_scaled, y_train)
val_predictions = grid_search.predict(X_val_scaled)

print("Best parameters:", grid_search.best_params_)
print("Best validation score:", grid_search.score(X_val_scaled, y_val))
print("Accuracy:", accuracy_score(y_val, val_predictions))

best_k = grid_search.best_params_['n_neighbors']
```

```
Best parameters: {'metric': 'manhattan', 'n_neighbors': 9}
Best validation score: 0.8099102947458351
Accuracy: 0.8099102947458351
```

In [386…
```
best_k
```

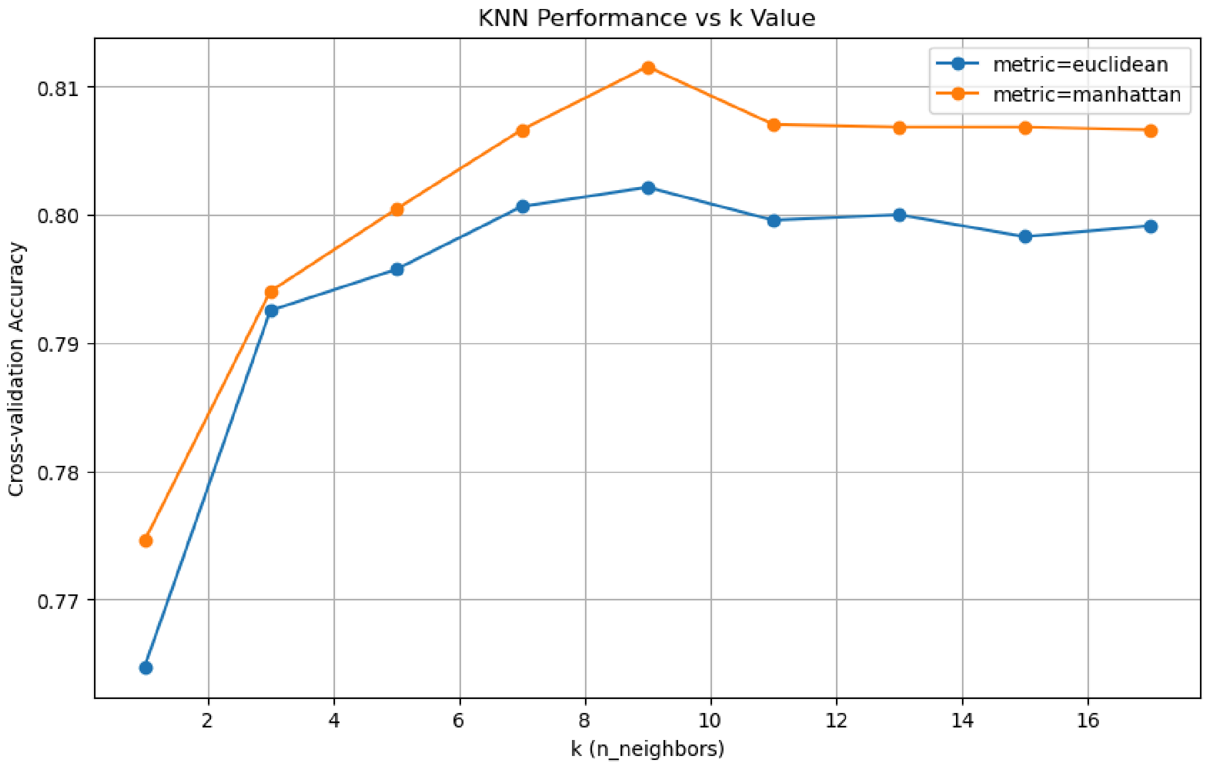
Out[386… 9

In [395…
```
k_values = param_grid['n_neighbors']
cv_results = grid_search.cv_results_

plt.figure(figsize=(10, 6))
for metric in param_grid['metric']:
    mask = cv_results['param_metric'] == metric
    plt.plot(k_values,
             cv_results['mean_test_score'][mask],
             marker='o',
             label=f'metric={metric}')

plt.xlabel('k (n_neighbors)')
plt.ylabel('Cross-validation Accuracy')
plt.title('KNN Performance vs k Value')
plt.legend()
```

```
plt.grid(True)
plt.show()
```



In [388…

```
help(KNeighborsClassifier)
```

```
Help on class KNeighborsClassifier in module sklearn.neighbors._classification:

class KNeighborsClassifier(sklearn.neighbors._base.KNeighborsMixin, sklearn.base.Cla
ssifierMixin, sklearn.neighbors._base.NeighborsBase)
 |  KNeighborsClassifier(n_neighbors=5, *, weights='uniform', algorithm='auto', leaf
_size=30, p=2, metric='minkowski', metric_params=None, n_jobs=None)
 |
 |  Classifier implementing the k-nearest neighbors vote.
 |
 |  Read more in the :ref:`User Guide <classification>`.
 |
 |  Parameters
 |  ----------
 |  n_neighbors : int, default=5
 |      Number of neighbors to use by default for :meth:`kneighbors` queries.
 |
 |  weights : {'uniform', 'distance'}, callable or None, 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.
 |
 |      Refer to the example entitled
 |      :ref:`sphx_glr_auto_examples_neighbors_plot_classification.py`
 |      showing the impact of the `weights` parameter on the decision
 |      boundary.
 |
 |  algorithm : {'auto', 'ball_tree', 'kd_tree', 'brute'}, default='auto'
 |      Algorithm used to compute the nearest neighbors:
 |
 |      - 'ball_tree' will use :class:`BallTree`
 |      - 'kd_tree' will use :class:`KDTree`
 |      - 'brute' will use a brute-force search.
 |      - 'auto' will attempt to decide the most appropriate algorithm
 |        based on the values passed to :meth:`fit` method.
 |
 |      Note: fitting on sparse input will override the setting of
 |      this parameter, using brute force.
 |
 |  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 : float, 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. This parameter is expecte
d
```

```
 |      to be positive.
 |
 |  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. See the
 |      documentation of `scipy.spatial.distance
 |      <https://docs.scipy.org/doc/scipy/reference/spatial.distance.html>`_ and
 |      the metrics listed in
 |      :class:`~sklearn.metrics.pairwise.distance_metrics` for valid metric
 |      values.
 |
 |      If metric is "precomputed", X is assumed to be a distance matrix and
 |      must be square during fit. X may be a :term:`sparse graph`, in which
 |      case only "nonzero" elements may be considered neighbors.
 |
 |      If metric is a callable function, it takes two arrays representing 1D
 |      vectors as inputs and must return one value indicating the distance
 |      between those vectors. This works for Scipy's metrics, but is less
 |      efficient than passing the metric name as a string.
 |
 |  metric_params : dict, default=None
 |      Additional keyword arguments for the metric function.
 |
 |  n_jobs : int, default=None
 |      The number of parallel jobs to run for neighbors search.
 |      ``None`` means 1 unless in a :obj:`joblib.parallel_backend` context.
 |      ``-1`` means using all processors. See :term:`Glossary <n_jobs>`
 |      for more details.
 |      Doesn't affect :meth:`fit` method.
 |
 |  Attributes
 |  ----------
 |  classes_ : array of shape (n_classes,)
 |      Class labels known to the classifier
 |
 |  effective_metric_ : str or callble
 |      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.
 |
 |  effective_metric_params_ : dict
 |      Additional keyword arguments for the metric function. For most metrics
 |      will be same with `metric_params` parameter, but may also contain the
 |      `p` parameter value if the `effective_metric_` attribute is set to
 |      'minkowski'.
 |
 |  n_features_in_ : int
 |      Number of features seen during :term:`fit`.
 |
 |      .. versionadded:: 0.24
 |
 |  feature_names_in_ : ndarray of shape (`n_features_in_`,)
 |      Names of features seen during :term:`fit`. Defined only when `X`
 |      has feature names that are all strings.
 |
 |      .. versionadded:: 1.0
```

```
 |  
 |  n_samples_fit_ : int
 |      Number of samples in the fitted data.
 |  
 |  outputs_2d_ : bool
 |      False when `y`'s shape is (n_samples, ) or (n_samples, 1) during fit
 |      otherwise True.
 |  
 |  See Also
 |  --------
 |  RadiusNeighborsClassifier: Classifier based on neighbors within a fixed radius.
 |  KNeighborsRegressor: Regression based on k-nearest neighbors.
 |  RadiusNeighborsRegressor: Regression based on neighbors within a fixed radius.
 |  NearestNeighbors: Unsupervised learner for implementing neighbor searches.
 |  
 |  Notes
 |  -----
 |  See :ref:`Nearest Neighbors <neighbors>` in the online documentation
 |  for a discussion of the choice of ``algorithm`` and ``leaf_size``.
 |  
 |  .. warning::
 |  
 |     Regarding the Nearest Neighbors algorithms, if it is found that two
 |     neighbors, neighbor `k+1` and `k`, have identical distances
 |     but different labels, the results will depend on the ordering of the
 |     training data.
 |  
 |  https://en.wikipedia.org/wiki/K-nearest_neighbor_algorithm
 |  
 |  Examples
 |  --------
 |  >>> X = [[0], [1], [2], [3]]
 |  >>> y = [0, 0, 1, 1]
 |  >>> from sklearn.neighbors import KNeighborsClassifier
 |  >>> neigh = KNeighborsClassifier(n_neighbors=3)
 |  >>> neigh.fit(X, y)
 |  KNeighborsClassifier(...)
 |  >>> print(neigh.predict([[1.1]]))
 |  [0]
 |  >>> print(neigh.predict_proba([[0.9]]))
 |  [[0.666... 0.333...]]
 |  
 |  Method resolution order:
 |      KNeighborsClassifier
 |      sklearn.neighbors._base.KNeighborsMixin
 |      sklearn.base.ClassifierMixin
 |      sklearn.neighbors._base.NeighborsBase
 |      sklearn.base.MultiOutputMixin
 |      sklearn.base.BaseEstimator
 |      sklearn.utils._estimator_html_repr._HTMLDocumentationLinkMixin
 |      sklearn.utils._metadata_requests._MetadataRequester
 |      builtins.object
 |  
 |  Methods defined here:
 |  
 |  __init__(self, n_neighbors=5, *, weights='uniform', algorithm='auto', leaf_size=
```

```
30, p=2, metric='minkowski', metric_params=None, n_jobs=None)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  fit(self, X, y)
 |      Fit the k-nearest neighbors classifier from the training dataset.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_samples, n_features) or
(n_samples, n_samples) if metric='precomputed'
 |          Training data.
 |
 |      y : {array-like, sparse matrix} of shape (n_samples,) or                 (n_
samples, n_outputs)
 |          Target values.
 |
 |      Returns
 |      -------
 |      self : KNeighborsClassifier
 |          The fitted k-nearest neighbors classifier.
 |
 |  predict(self, X)
 |      Predict the class labels for the provided data.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_queries, n_features),
or (n_queries, n_indexed) if metric == 'precomputed'
 |          Test samples.
 |
 |      Returns
 |      -------
 |      y : ndarray of shape (n_queries,) or (n_queries, n_outputs)
 |          Class labels for each data sample.
 |
 |  predict_proba(self, X)
 |      Return probability estimates for the test data X.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_queries, n_features),
or (n_queries, n_indexed) if metric == 'precomputed'
 |          Test samples.
 |
 |      Returns
 |      -------
 |      p : ndarray of shape (n_queries, n_classes), or a list of n_outputs
of such arrays if n_outputs > 1.
 |          The class probabilities of the input samples. Classes are ordered
 |          by lexicographic order.
 |
 |  set_score_request(self: sklearn.neighbors._classification.KNeighborsClassifier,
*, sample_weight: Union[bool, NoneType, str] = '$UNCHANGED$') -> sklearn.neighbors._
classification.KNeighborsClassifier from sklearn.utils._metadata_requests.RequestMet
hod.__get__.<locals>
 |      Request metadata passed to the ``score`` method.
```

```
 |
 |      Note that this method is only relevant if
 |      ``enable_metadata_routing=True`` (see :func:`sklearn.set_config`).
 |      Please see :ref:`User Guide <metadata_routing>` on how the routing
 |      mechanism works.
 |
 |      The options for each parameter are:
 |
 |      - ``True``: metadata is requested, and passed to ``score`` if provided. The
request is ignored if metadata is not provided.
 |
 |      - ``False``: metadata is not requested and the meta-estimator will not pass
it to ``score``.
 |
 |      - ``None``: metadata is not requested, and the meta-estimator will raise an
error if the user provides it.
 |
 |      - ``str``: metadata should be passed to the meta-estimator with this given a
lias instead of the original name.
 |
 |      The default (``sklearn.utils.metadata_routing.UNCHANGED``) retains the
 |      existing request. This allows you to change the request for some
 |      parameters and not others.
 |
 |      .. versionadded:: 1.3
 |
 |      .. note::
 |          This method is only relevant if this estimator is used as a
 |          sub-estimator of a meta-estimator, e.g. used inside a
 |          :class:`~sklearn.pipeline.Pipeline`. Otherwise it has no effect.
 |
 |      Parameters
 |      ----------
 |      sample_weight : str, True, False, or None,                     default=sklea
rn.utils.metadata_routing.UNCHANGED
 |          Metadata routing for ``sample_weight`` parameter in ``score``.
 |
 |      Returns
 |      -------
 |      self : object
 |          The updated object.
 |
 |  ----------------------------------------------------------------------
 |  Data and other attributes defined here:
 |
 |  __abstractmethods__ = frozenset()
 |
 |  __annotations__ = {'_parameter_constraints': <class 'dict'>}
 |
 |  __slotnames__ = []
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.neighbors._base.KNeighborsMixin:
 |
 |  kneighbors(self, X=None, n_neighbors=None, return_distance=True)
 |      Find the K-neighbors of a point.
```

```
 |
 |      Returns indices of and distances to the neighbors of each point.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix}, shape (n_queries, n_features),
or (n_queries, n_indexed) if metric == 'precomputed', default=None
 |          The query point or points.
 |          If not provided, neighbors of each indexed point are returned.
 |          In this case, the query point is not considered its own neighbor.
 |
 |      n_neighbors : int, default=None
 |          Number of neighbors required for each sample. The default is the
 |          value passed to the constructor.
 |
 |      return_distance : bool, default=True
 |          Whether or not to return the distances.
 |
 |      Returns
 |      -------
 |      neigh_dist : ndarray of shape (n_queries, n_neighbors)
 |          Array representing the lengths to points, only present if
 |          return_distance=True.
 |
 |      neigh_ind : ndarray of shape (n_queries, n_neighbors)
 |          Indices of the nearest points in the population matrix.
 |
 |      Examples
 |      --------
 |      In the following example, we construct a NearestNeighbors
 |      class from an array representing our data set and ask who's
 |      the closest point to [1,1,1]
 |
 |      >>> samples = [[0., 0., 0.], [0., .5, 0.], [1., 1., .5]]
 |      >>> from sklearn.neighbors import NearestNeighbors
 |      >>> neigh = NearestNeighbors(n_neighbors=1)
 |      >>> neigh.fit(samples)
 |      NearestNeighbors(n_neighbors=1)
 |      >>> print(neigh.kneighbors([[1., 1., 1.]]))
 |      (array([[0.5]]), array([[2]]))
 |
 |      As you can see, it returns [[0.5]], and [[2]], which means that the
 |      element is at distance 0.5 and is the third element of samples
 |      (indexes start at 0). You can also query for multiple points:
 |
 |      >>> X = [[0., 1., 0.], [1., 0., 1.]]
 |      >>> neigh.kneighbors(X, return_distance=False)
 |      array([[1],
 |             [2]]...)
 |
 |  kneighbors_graph(self, X=None, n_neighbors=None, mode='connectivity')
 |      Compute the (weighted) graph of k-Neighbors for points in X.
 |
 |      Parameters
 |      ----------
 |      X : {array-like, sparse matrix} of shape (n_queries, n_features),
```

```
        or (n_queries, n_indexed) if metric == 'precomputed', default=None
 |              The query point or points.
 |              If not provided, neighbors of each indexed point are returned.
 |              In this case, the query point is not considered its own neighbor.
 |              For ``metric='precomputed'`` the shape should be
 |              (n_queries, n_indexed). Otherwise the shape should be
 |              (n_queries, n_features).
 |
 |          n_neighbors : int, default=None
 |              Number of neighbors for each sample. The default is the value
 |              passed to the constructor.
 |
 |          mode : {'connectivity', 'distance'}, default='connectivity'
 |              Type of returned matrix: 'connectivity' will return the
 |              connectivity matrix with ones and zeros, in 'distance' the
 |              edges are distances between points, type of distance
 |              depends on the selected metric parameter in
 |              NearestNeighbors class.
 |
 |          Returns
 |          -------
 |          A : sparse-matrix of shape (n_queries, n_samples_fit)
 |              `n_samples_fit` is the number of samples in the fitted data.
 |              `A[i, j]` gives the weight of the edge connecting `i` to `j`.
 |              The matrix is of CSR format.
 |
 |          See Also
 |          --------
 |          NearestNeighbors.radius_neighbors_graph : Compute the (weighted) graph
 |              of Neighbors for points in X.
 |
 |          Examples
 |          --------
 |          >>> X = [[0], [3], [1]]
 |          >>> from sklearn.neighbors import NearestNeighbors
 |          >>> neigh = NearestNeighbors(n_neighbors=2)
 |          >>> neigh.fit(X)
 |          NearestNeighbors(n_neighbors=2)
 |          >>> A = neigh.kneighbors_graph(X)
 |          >>> A.toarray()
 |          array([[1., 0., 1.],
 |                 [0., 1., 1.],
 |                 [1., 0., 1.]])
 |
 |      ----------------------------------------------------------------------
 |      Data descriptors inherited from sklearn.neighbors._base.KNeighborsMixin:
 |
 |      __dict__
 |          dictionary for instance variables
 |
 |      __weakref__
 |          list of weak references to the object
 |
 |      ----------------------------------------------------------------------
 |      Methods inherited from sklearn.base.ClassifierMixin:
 |
```

```
 |  score(self, X, y, sample_weight=None)
 |      Return the mean accuracy on the given test data and labels.
 |
 |      In multi-label classification, this is the subset accuracy
 |      which is a harsh metric since you require for each sample that
 |      each label set be correctly predicted.
 |
 |      Parameters
 |      ----------
 |      X : array-like of shape (n_samples, n_features)
 |          Test samples.
 |
 |      y : array-like of shape (n_samples,) or (n_samples, n_outputs)
 |          True labels for `X`.
 |
 |      sample_weight : array-like of shape (n_samples,), default=None
 |          Sample weights.
 |
 |      Returns
 |      -------
 |      score : float
 |          Mean accuracy of ``self.predict(X)`` w.r.t. `y`.
 |
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.base.BaseEstimator:
 |
 |  __getstate__(self)
 |      Helper for pickle.
 |
 |  __repr__(self, N_CHAR_MAX=700)
 |      Return repr(self).
 |
 |  __setstate__(self, state)
 |
 |  __sklearn_clone__(self)
 |
 |  get_params(self, deep=True)
 |      Get parameters for this estimator.
 |
 |      Parameters
 |      ----------
 |      deep : bool, default=True
 |          If True, will return the parameters for this estimator and
 |          contained subobjects that are estimators.
 |
 |      Returns
 |      -------
 |      params : dict
 |          Parameter names mapped to their values.
 |
 |  set_params(self, **params)
 |      Set the parameters of this estimator.
 |
 |      The method works on simple estimators as well as on nested objects
 |      (such as :class:`~sklearn.pipeline.Pipeline`). The latter have
 |      parameters of the form ``<component>__<parameter>`` so that it's
```

```
 |      possible to update each component of a nested object.
 |      
 |      Parameters
 |      ----------
 |      **params : dict
 |          Estimator parameters.
 |      
 |      Returns
 |      -------
 |      self : estimator instance
 |          Estimator instance.
 |  
 |  ----------------------------------------------------------------------
 |  Methods inherited from sklearn.utils._metadata_requests._MetadataRequester:
 |  
 |  get_metadata_routing(self)
 |      Get metadata routing of this object.
 |      
 |      Please check :ref:`User Guide <metadata_routing>` on how the routing
 |      mechanism works.
 |      
 |      Returns
 |      -------
 |      routing : MetadataRequest
 |          A :class:`~sklearn.utils.metadata_routing.MetadataRequest` encapsulating
 |          routing information.
 |  
 |  ----------------------------------------------------------------------
 |  Class methods inherited from sklearn.utils._metadata_requests._MetadataRequeste
r:
 |  
 |  __init_subclass__(**kwargs)
 |      Set the ``set_{method}_request`` methods.
 |      
 |      This uses PEP-487 [1]_ to set the ``set_{method}_request`` methods. It
 |      looks for the information available in the set default values which are
 |      set using ``__metadata_request__*`` class attributes, or inferred
 |      from method signatures.
 |      
 |      The ``__metadata_request__*`` class attributes are used when a method
 |      does not explicitly accept a metadata through its arguments or if the
 |      developer would like to specify a request value for those metadata
 |      which are different from the default ``None``.
 |      
 |      References
 |      ----------
 |      .. [1] https://www.python.org/dev/peps/pep-0487
```

In [389…

```
best_knn = KNeighborsClassifier(n_neighbors=best_k,metric=best_params['metric'])
best_knn.fit(X_train_scaled, y_train)
```

Out[389…

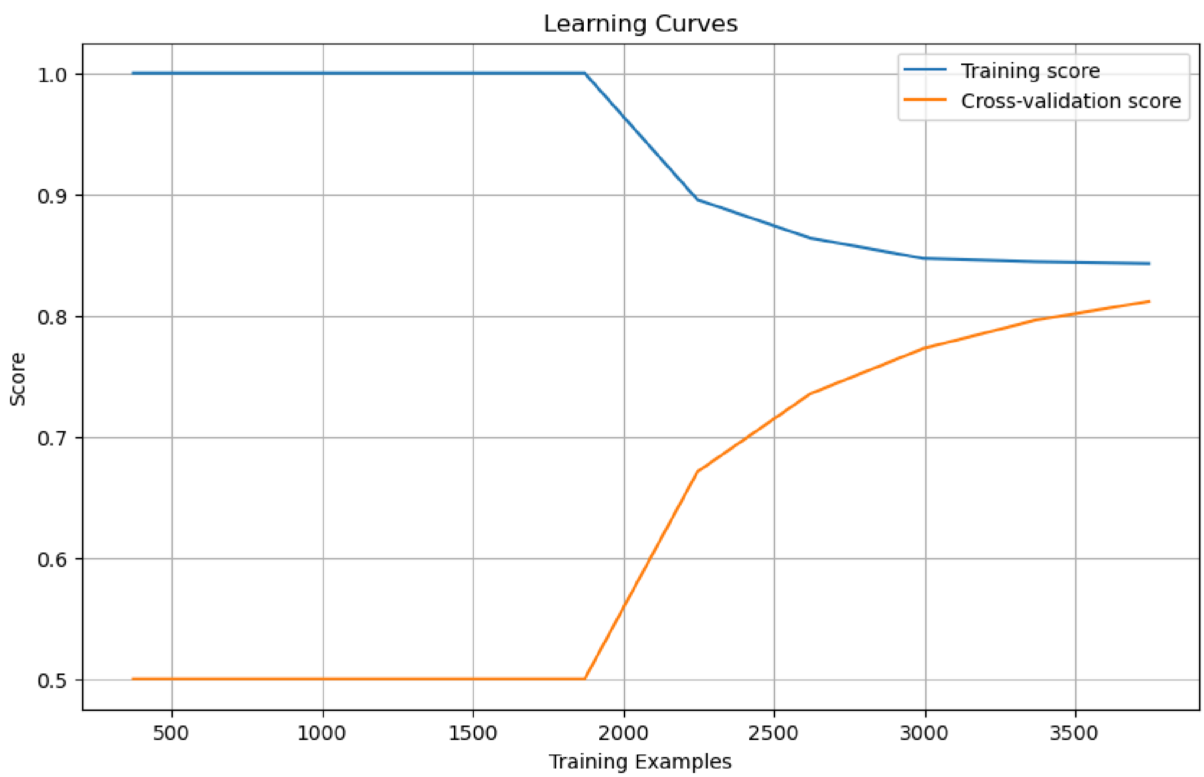
KNeighborsClassifier

```
KNeighborsClassifier(metric='manhattan', n_neighbors=9)
```

In [390…

```
train_sizes, train_scores, val_scores = learning_curve(
    best_knn, X_train_scaled, y_train,
    train_sizes=np.linspace(0.1, 1.0, 10),
    cv=5, n_jobs=-1)

plt.figure(figsize=(10, 6))
plt.plot(train_sizes, train_scores.mean(axis=1), label='Training score')
plt.plot(train_sizes, val_scores.mean(axis=1), label='Cross-validation score')
plt.xlabel('Training Examples')
plt.ylabel('Score')
plt.title('Learning Curves')
plt.legend(loc='best')
plt.grid(True)
plt.show()
```



In [391…

```
y_test_pred = best_knn.predict(X_test_scaled)
```

In [392…

```
accuracy = accuracy_score(y_test, y_test_pred)
precision = precision_score(y_test, y_test_pred, average='weighted')
recall = recall_score(y_test, y_test_pred, average='weighted')
f1 = f1_score(y_test, y_test_pred, average='weighted')
print(f"Accuracy: {accuracy}")
print(f"Precision: {precision}")
print(f"Recall: {recall}")
print(f"F1-score: {f1}")
```

```
Accuracy: 0.8074240159441953
Precision: 0.8190425520951419
Recall: 0.8074240159441953
F1-score: 0.8056546528713431
```

In [393…

```
cm = confusion_matrix(y_test, y_test_pred)

plt.figure(figsize=(8, 6))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.show()
```