Nearest Neighbor Search through Cover Trees

This class provides functionality for unsupervised nearest neighbor search, which is the foundation of many other learning methods, notably manifold learning and spectral clustering.

The goal is to find a number of points from the given database closest in distance to the query point. The distance can be, in general, any metric measure: standard Euclidean distance is the most common choice and the one currently implemented. In future, adding other metrics should not be too difficult.

Standard packages, like those in scikit learn use KD-tree or Ball trees, which do not scale very well, especially with respect to dimension. For example, Ball Trees of scikit learn takes $O(n^2)$ construction time and a search query can be linear in worst case making it no better than brute force in some cases.

In this class, we implement a modified version of the Cover Tree data structure that allow fast retrieval in logarithmic time. The key properties are: O(n log n) construction time, O(log n) retrieval, and polynomial dependence on the expansion constant of the underlying space. In addition, it allows insertion and removal of points in database. *The class is pickle-able.*

Construction functions

CoverTree.from_matrix(*points***)**

Constructs a new cover tree on a given collection of points. The function utilizes multiple cores.

Parameters:	**point***: 2D ndarray* A contiguous numpy array of shape (num_ points, dim)..
Returns:	**Cover_tree***: CoverTree* Returns an object of class CoverTree containing the points arranged in the tree.

Manipulation functions

CoverTree.insert(*point***)**

Inserts a new point into the cover tree. [Thread-safe]

Parameters:	**point***: 1D ndarray* A contiguous numpy array of same dimension as other points.
Returns:	**success***: Boolean*

	Returns true is the point was successfully inserted. If the point to be inserted is a duplicate one, i.e. already present in the tree then it is not inserted and a false is returned.

CoverTree.remove(*point***)**

Removes a point from the cover tree. [NOT Thread-safe]

Parameters:	**point***: 1D ndarray* A contiguous numpy array of same dimension as other points.
Returns:	**success***: Boolean* Returns true is the point was successfully removed. If the point to be removed is not found in the tree, then a false is returned. Also if the point to be removed is the root of the tree a false is returned with sorry message that efficient way to delete root node is not known.

Search functions

CoverTree.NearestNeighbour(*points***)**

Searches for the nearest neighbour in the cover tree for each point. [Thread-safe]

Parameters:	**points***: 2D ndarray* A contiguous numpy array of shape (num_query_points, dim).
Returns:	**neighbours***: 2D ndarray* Returns the nearest neighbour for each of the point. Same shape as points.

CoverTree.kNearestNeighbour(*points, k=10***)**

Searches for the k nearest neighbours in the cover tree for each point. [Thread-safe]

Parameters:	**points***: 2D ndarray* A contiguous numpy array of shape (num_query_points, dim). **k:** *integer* The number of nearest numbers to return.
Returns:	**neighbours***: 3D ndarray*

	Returns the k nearest neighbours for each of the point. Shape is (num_query_points, k, dim)

CoverTree.range(*points, r=1.0***)**

Searches for all the entries in the cover tree, which are at most *r* distance from *points*. [Thread-safe]

Parameters:	**points***: 2D ndarray* A contiguous numpy array of shape (num_query_points, dim). **r:** *floating-point number* The distance radius for which to find points.
Returns:	**neighbours***: List of 2D ndarray* Returns all the entries in the cover tree which are at most *r* distance for each of the point. Shape is (k, dim)* num_query_points

Example

```
import CoverTree
import pickle
import numpy as np

# Generate random points
x = np.random.rand(10000,16)
y = np.random.rand(2000,16)

# Build cover tree from numpy matrix
ct = CoverTree.from_matrix(x)

# Search for nearest neighbors
a = ct.NearestNeighbour(y)
b = ct.kNearestNeighbours(y,3)

# Pickle the cover tree for future searches
with open('tree.dat', 'wb') as f:
    pickle.dump(ct, f)

with open('tree.dat', 'rb') as f:
   ct_new = pickle.load(f)
```

Clustering through K-Means

This class provides functionality for unsupervised clustering, which according to Wikipedia is “the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar to each other than to those in other groups”. It is a main task of exploratory data mining, and a common technique for statistical data analysis.

The similarity measure can be, in general, any metric measure: standard Euclidean distance is the most common choice and the one currently implemented. In future, adding other metrics should not be too difficult.

Standard packages, like those in scikit learn run on a single machine and often only on one thread. Whereas our underlying C++ implementation can be distributed to run on multiple machines. To enable the distribution through python interface is work in progress.

In this class, we implement a K-Means clustering using Llyod’s algorithm and speed-up using Cover Trees. The API is similar to `sklearn.cluster.KMeans`. *The class is pickle-able.*

KMeans.init(*k, iters, init='covertree'***)**

Configure an instance of K-Means clustering task. The function utilizes multiple cores.

Parameters:	**k***: integer* The number of clusters to form as well as the number of centroids to generate. **iters***: integer* The number of iterations of the Lloyd’s algorithm for K-Means clustering. **init***: {'random', 'kmeanspp', 'covertree', or an ndarray}* ‘random’: choose k observations (rows) at random from data for the initial centroids. ‘kmeanspp’ : selects initial cluster centers by finding well spread out points using cover trees to speed up convergence. ‘covertree’ : selects initial cluster centers by sampling to speed up convergence. If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers.
Returns:	**kmeans***: KMeans* Returns an object of class KMeans configured according to the supplied parameters.

KMeans.fit(*training_points, [validation_points]***)**

Compute k-means clustering.

Parameters:	**training_points***: 2D ndarray* A contiguous numpy array of shape (num_points, dim) that represents training instances to cluster. **validation_points***: 2D ndarray* A contiguous numpy array of shape (num_points, dim) which represents validation instances to validate the clustering results learned after each iteration of the Lloyd's algorithm.
Returns:	**score***: floating-point number* Returns the final training/validation score (-ve of K-Means objective value).

KMeans.evaluate(*test_points***)**

Finds the score of learned model on a set of test points.

Parameters:	**test_points***: 2D ndarray* A contiguous numpy array of same dimension as other points that represents test instances to test the clustering results.
Returns:	**score***: floating-point number* Returns the test score (-ve of K-Means objective value).

KMeans.predict(*test_points***)**

Finds the closest cluster for the given set of test points using the learned model.

Parameters:	**test_points***: 2D ndarray* A contiguous numpy array of shape (num_points, dim).
Returns:	**cluster_assignments***: 1D ndarray* Returns the index of the cluster each sample belongs to.

KMeans.get_centers()

Get current cluster centers for this model.

Parameters:	-
Returns:	**cluster_centres***: 2D ndarray* Returns a numpy array of shape (k, dim) containing the cluster centers.

Example

```
import KMeans
import pickle
import numpy as np

# Generate random points
x = np.random.rand(10000,16)
y = np.random.rand(2000,16)

# Configure K-Means instance
ctm = Clustering.init(100, 10, 'covertree', x)

# Cluster the points
ctm.fit(x,y)

# Get the cluster centers
cc = ctm.get_centers(x,y)

# Get the cluster assignments
z = ctm.predict(y)

# Pickle the inferred model for future use
with open('kmeans.dat', 'wb') as f:
    pickle.dump(ctm, f)

with open('kmeans.dat', 'rb') as f:
   ctm_new = pickle.load(f)
```

Gaussian Mixture Models

This class provides functionality for unsupervised inference on Gaussian mixture model, which is a probabilistic model that assumes all the data points are generated from a mixture of a finite number of Gaussian distributions with unknown parameters. It can be viewed as a generalization of the K-Means clustering to incorporate information about the covariance structure of the data.

Standard packages, like those in scikit learn run on a single machine and often only on one thread. Whereas our underlying C++ implementation can be distributed to run on multiple machines. To enable the distribution through python interface is work in progress.

In this class, we implement inference on (Bayesian) Gaussian mixture models using Canopy algorithm. The API is similar to `sklearn.mixture.GaussianMixture`. *The class is pickle-able.*

GMM.init(*k, iters, init_mean='covertree', init_var=None***)**

Configure an instance of GMM clustering task. The function utilizes multiple cores.

Parameters:	**k***: integer* The number of clusters to form as well as the number of centroids to generate. **iters***: integer* The number of iterations of inference. **init_mean***: {'random', 'kmeanspp', 'covertree', or an ndarray}* 'random': choose k observations (rows) at random from data for the initial centroids. 'kmeanspp' : selects initial cluster centers by finding well spread out points using cover trees to speed up convergence. 'covertree' : selects initial cluster centers by sampling to speed up convergence. If an ndarray is passed, it should be of shape (n_clusters, n_features) and gives the initial centers. **Init_var***: { an ndarray or None}* User supplied initialization of variances of mixture components. It can be 2d numpy array of shape (k, dim) or an 1d numpy array of shape (dim) which will be replicated for all components. If not provided, then global variance of the training dataset is used.
Returns:	**gmm***: GMM*

	Returns an object of class GMM configured according to the supplied parameters.

GMM.fit(*training_points, [validation_points]***)**

Inference on the Gaussian mixture model.

Parameters:	**training_points***: 2D ndarray* A contiguous numpy array of shape (num_points, dim) that represents training instances to cluster. **validation_points***: 2D ndarray* A contiguous numpy array of shape (num_points, dim) which represents validation instances to validate the clustering results learned after each iteration of ESCA algorithm.
Returns:	**score***: floating-point number* Returns the final training/validation log likelihood.

GMM.evaluate(*test_points***)**

Finds the log likelihood of learned model on a set of test points.

Parameters:	**test_points***: 2D ndarray* A contiguous numpy array of same dimension as other points that represents test instances to test the clustering results.
Returns:	**score***: floating-point number* Returns the test log likelihood.

GLDA.predict(*test_points***)**

Finds the closest cluster for the given set of test points using the learned model.

Parameters:	**test_points***: 2D ndarray* A contiguous numpy array of shape (num_points, dim).
Returns:	**cluster_assignments***: 1D ndarray* Returns the index of the cluster each sample belongs to.

GLDA.get_centers()

Get current cluster centers and variances for this model.

Parameters:	-
	cluster_centres: *2D ndarray, 2d ndarry*
Returns:	Returns two numpy array, each of shape (k, dim) containing the cluster centers and variances respectively.

Example

```
import GMM
import pickle
import numpy as np

# Generate random points
x = np.random.rand(10000,16)
y = np.random.rand(2000,16)

# Configure GMM instance
ctm = GMM.init(100, 10, 'covertree', x)

# Inference on the points
ctm.fit(x,y)

# Get the cluster centers
cc = ctm.get_centers(x,y)

# Get the cluster assignments
z = ctm.predict(y)

# Pickle the inferred model for future use
with open(gmm.dat', 'wb') as f:
    pickle.dump(ctm, f)

with open(gmm.dat', 'rb') as f:
   ctm_new = pickle.load(f)
```

Latent Dirichlet Allocation

This class provides functionality for unsupervised inference on latent Dirichlet allocation, which is a probabilistic topic model of corpora of documents which seeks to represent the underlying thematic structure of the document collection. They have emerged as a powerful new technique of finding useful structure in an unstructured collection as it learns distributions over words. The high probability words in each distribution gives us a way of understanding the contents of the corpus at a very high level. In LDA, each document of the corpus is assumed to have a distribution over K topics, where the discrete topic distributions are drawn from a symmetric dirichlet distribution.

Standard packages, like those in scikit learn are inefficient in addition to being limited to a single machine. Whereas our underlying C++ implementation can be distributed to run on multiple machines. To enable the distribution through python interface is work in progress.

In this class, we implement inference on latent Dirichlet Allocation using ESCA algorithm. The API is similar to `sklearn.decomposition.LatentDirichletAllocation`. *The class is pickle-able.*

LDA.init(*k, iters, vocab***)**

Configure an instance of LDA task. The function utilizes multiple cores.

Parameters:	**k***: integer* The number of topics. **iters***: integer* The number of iterations of inference. **vocab***: list of strings* The list of words in the corpus acting as a mapping from word ids (integers) to words (strings). It is used to determine the vocabulary size, as well as for debugging and topic printing.
Returns:	**lda***: LDA* Returns an object of class LDA configured according to the supplied parameters.

LDA.fit(*training_docs, [validation_docs]***)**

Inference on the latent Dirichlet allocation model.

Parameters:	**training_docs***: list of 1D ndarray*

	A list of 1d numpy array of dtype uint32. Each numpy array contains a document with each token mapped to its word id. **validation_docs***: list of 1D ndarray* A list of 1d numpy array of dtype uint32. Each numpy array contains a document with each token mapped to its word id. This represents validation docs to validate the results learned after each iteration of canopy algorithm.
Returns:	**score***: floating-point number* Returns the final per-token training/validation log likelihood (-ve log perplexity)

LDA.evaluate(*test_docs***)**

Finds the per-token log likelihood (-ve log perplexity) of learned model on a set of test docs.

Parameters:	**test_docs***: list of 1D ndarray* A list of 1d numpy array of dtype uint32. Each numpy array contains a document with each token mapped to its word id. This represents test docs to test the learned model.
Returns:	**score***: floating-point number* Returns the final per-token log likelihood (-ve log perplexity).

LDA.predict(*test_points***)**

Finds the token topic assignment (and consequently topic-per-document distribution) for the given set of docs using the learned model.

Parameters:	**test_docs***: list of 1D ndarray* A list of 1d numpy array of dtype uint32. Each numpy array contains a document with each token mapped to its word id.
Returns:	**topic_assignments***: list of 1D ndarray* Returns the index of the topic each token belongs to.

LDA.get_topic_matrix()

Get current word|topic distribution matrix for this model.

Parameters:	-

Returns:	**topic_matrix***: 2D ndarray* Returns a numpy array of shape (vocab_size,k) with each column containing the word|topic distribution.

LDA.get_top_words(*num_top=15***)**

Get the top words of each topic for this model.

Parameters:	**num_top***: integer* The number of top words requested.
Returns:	**top_words***: list of list of strings* Returns a list of size k containing list of size num_top words.

Example

```
import LDA
import pickle
import numpy as np

# Load NIPS data
trngdata, vocab = read_corpus('data/nips.train')
testdata, vocab = read_corpus('data/nips.test', vocab)

# Configure LDA instance
clda = LDA.init(10, 10, vocab)

# Inference on the docs
clda.fit(trngdata, testdata)

# Get top words
tpw = clda.get_top_words()

# Test on held out data using learned model
S = clda.evaluate(testdata)

# Pickle the inferred model for future use
with open(lda.dat', 'wb') as f:
    pickle.dump(clda, f)

with open(lda.dat', 'rb') as f:
   clda_new = pickle.load(f)
```

Gaussian Latent Dirichlet Allocation

This class provides functionality for unsupervised inference on Gaussian latent Dirichlet allocation, which replace LDA's parameterization of "topics" as categorical distributions over opaque word types with multivariate Gaussian distributions on the embedding space. This encourages the model to group words that are a priori known to be semantically related into topics, as continuous space word embeddings learned from large, unstructured corpora have been shown to be effective at capturing semantic regularities in language.

Using vectors learned from a domain-general corpus (e.g. English Wikipedia), qualitatively, Gaussian LDA infers different (but still very sensible) topics relative to standard LDA. Quantitatively, the technique outperforms existing models at dealing with OOV words in held-out documents.

No standard packages exists. Our underlying C++ implementation can be distributed to run on multiple machines. To enable the distribution through python interface is work in progress.

In this class, we implement inference on Gaussian latent Dirichlet Allocation using Canopy algorithm. In case of full covariance matrices, it exploits the Cholesky decompositions of covariance matrices of the posterior predictive distributions and performs efficient rank-one updates. The API is similar to `sklearn.decomposition.LatentDirichletAllocation`. *The class is pickle-able.*

GLDA.init(*k, iters, vocab, vectors***)**

Configure an instance of GLDA task. The function utilizes multiple cores.

Parameters:	**k***: integer* The number of topics. **iters***: integer* The number of iterations of inference. **vocab***: list of strings* The list of words in the corpus acting as a mapping from word ids (integers) to words (strings). It is used to determine the vocabulary size, as well as for debugging and topic printing. **vectors***: 2D ndarray* A contiguous numpy array of shape (vocab_size, dim) containing the continuous word embeddings. The order of the vectors should match the order of words in the vocab.
Returns:	**glda***: GLDA*

	Returns an object of class GLDA configured according to the supplied parameters.

GLDA.fit(*training_docs, [validation_docs]***)**

Inference on the Gaussian latent Dirichlet allocation model.

Parameters:	**training_docs***: list of 1D ndarray* A list of 1d numpy array of dtype uint32. Each numpy array contains a document with each token mapped to its word id. **validation_docs***: list of 1D ndarray* A list of 1d numpy array of dtype uint32. Each numpy array contains a document with each token mapped to its word id. This represents validation docs to validate the results learned after each iteration of canopy algorithm.
Returns:	**score***: floating-point number* Returns the final per-token training/validation log likelihood (-ve log perplexity)

GLDA.evaluate(*test_docs***)**

Finds the per-token log likelihood (-ve log perplexity) of learned model on a set of test docs.

Parameters:	**test_docs***: list of 1D ndarray* A list of 1d numpy array of dtype uint32. Each numpy array contains a document with each token mapped to its word id. This represents test docs to test the learned model.
Returns:	**score***: floating-point number* Returns the final per-token log likelihood (-ve log perplexity).

GLDA.predict(*test_points***)**

Finds the token topic assignment (and consequently topic-per-document distribution) for the given set of docs using the learned model.

Parameters:	**test_docs***: list of 1D ndarray* A list of 1d numpy array of dtype uint32. Each numpy array contains a document with each token mapped to its word id.
Returns:	**topic_assignments***: list of 1D ndarray*

	Returns the index of the topic each token belongs to.

GLDA.get_topic_matrix()

Get current topic centers and variances for this model.

Parameters:	-
Returns:	**topic_centres***: 2D ndarray, 2d ndarry* Returns two numpy array, each of shape (k, dim) containing the topic centers and variances respectively.

GLDA.get_top_words(*num_top=15***)**

Get the top words of each topic for this model.

Parameters:	**num_top***: integer* The number of top words requested.
Returns:	**top_words***: list of list of strings* Returns a list of size k containing list of size num_top words.

Example

```
import GLDA
import pickle
import numpy as np

# Load NIPS data
trngdata, vocab = read_corpus('data/nips.train')
testdata, vocab = read_corpus('data/nips.test', vocab)
vectors = get_embedding('data/word.emb', vocab)

# Configure GLDA instance
cglda = GLDA.init(10, 10, vocab, vectors)

# Inference on the docs
cglda.fit(trngdata, testdata)

# Get top words
tpw = cglda.get_top_words()

# Test on held out data using learned model
S = cglda.evaluate(testdata)
```

```
# Pickle the inferred model for future use
with open(glda.dat', 'wb') as f:
    pickle.dump(cglda, f)

with open(glda.dat', 'rb') as f:
   cglda_new = pickle.load(f)
```