emlens

Release 0.0.1

Sadamori Kojaku

Oct 10, 2021

CONTENTS:

Embedding is a potent to tool for generating a vector representation of words and networks. To fully leverage its utility, the embedding space needs to be high dimensional. While computers have no problem in understanding high dimensional space, our brains do not. This `emlens` aims to fill the gap by providing a set of tools for visualizing the embedding space, quantifying correlation to metadata, and calculating densities.

This package is under active development. If you have issues and feature requests, please raise them through Github.

CHAPTER

ONE

EMLENS

1.1 emlens package

1.1.1 Submodules

1.1.2 emlens.density_estimation module

emlens.density_estimation.**estimate_pdf**(*target*, *emb*, *C=0.1*)

Estimate the density of entities at the given target locations in the embedding space using the density estimator based on the k-nearest neighbors.

Parameters

- **target** (*numpy.array, shape=(num_target, dim)*) – Target location at which the density is calculated.
- **emb** (*numpy.ndarray, (num_entities, dim)*) – Embedding vectors for the entities
- **C** (*float, optional*) – Bandwidth for kernels. Ranges between (0,1]. Roughly C * num_entities nearest neighbors will be used for estimating the density at a single target location.

Returns Log-density of points at the target locations.

Return type numpy.ndarray (num_target,)

Reference https://faculty.washington.edu/yenchic/18W_425/Lec7_knn_basis.pdf

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20)
>>> target = np.random.randn(10, 20)
>>> density = emlens.estimate_pdf(target=target, emb = emb)
```

1.1.3 emlens.metrics module

emlens.metrics.**assortativity**(*emb*, *y*, *k=5*, *metric='euclidean'*, *gpu_id=None*)

Calculate the assortativity of *y* for close entities in the embedding space. A positive/negative assortativity indicates that the close entities tend to have a similar/dissimilar *y*. Zero assortativity means *y* is independent of the embedding.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **y** (*numpy.ndarray (num_entities,)*) – feature of y
- **A** (*scipy.csr_matrix, optional*) – precomputed adjacency matrix of the graph. If None, a k-nearest neighbor graph will be constructed, defaults to None
- **k** (*int or list, optional*) – Number of the nearest neighbors. If a list is given, the assortativity for each k in the list will be calculated (in the same order of the list), defaults to 5

Paramm metric Distance metric for finding nearest neighbors. Available metric *metric="euclidean"*, *metric="cosine"* , *metric="dotsim"*

Returns assortativity

Return type float

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20)
>>> y = np.random.randn(emb.shape[0])
>>> rho = emlens.assortativity(emb, y)
```

Note: To calculate the assortativity, a k-nearest neighbor graph is constructed. Then, the assortativity is calculated as the Pearson correlation of y between the adjacent nodes.

emlens.metrics.**effective_dimension**(*emb*, *q=1*, *normalize=False*, *is_cov=False*)

Effective dimensionality of a set of points in space.

Effection dimensionality is the number of orthogonal dimensions needed to capture the overall correlational structure of data. See Del Giudice, M. (2020). Effective Dimensionality: A Tutorial. _Multivariate Behavioral Research, 0(0), 1–16. https://doi.org/10.1080/00273171.2020.1743631.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **q** (*int, optional*) – Parameter for the Renyi entropy function, defaults to 1
- **normalize** (*bool, optional*) – Set True to center data. For spherical or quasi-spherical data (such as the embedding by word2vec), normalize=False is recommended, defaults to False
- **is_cov** (*bool, optional*) – Set True if *emb* is the covariance matrix, defaults to False

Returns effective dimensionality

Return type float

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20)
>>> ed = emlens.effective_dimension(emb)
```

emlens.metrics.**effective_dimension_vector**(*emb*, *normalize=False*, *is_cov=False*)

Effective dimensionality of a set of points in space.

Effection dimensionality is the number of orthogonal dimensions needed to capture the overall correlational structure of data. See Del Giudice, M. (2020). Effective Dimensionality: A Tutorial. _Multivariate Behavioral Research, 0(0), 1–16. https://doi.org/10.1080/00273171.2020.1743631.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **q** (*int, optional*) – Parameter for the Renyi entropy function, defaults to 1
- **normalize** (*bool, optional*) – Set True to center data. For spherical or quasi-spherical data (such as the embedding by word2vec), normalize=False is recommended, defaults to False
- **is_cov** (*bool, optional*) – Set True if *emb* is the covariance matrix, defaults to False

Returns effective dimensionality

Return type float

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20)
>>> ed = emlens.effective_dimension(emb)
```

emlens.metrics.**element_sim**(*emb*, *group_ids*, *A=None*, *k=10*, *metric='euclidean'*, *gpu_id=None*)

Calculate the Element Centric Clustering Similarity for the entities with group membership.

Gates, A. J., Wood, I. B., Hetrick, W. P., & Ahn, Y. Y. (2019). Element-centric clustering comparison unifies overlaps and hierarchy. Scientific Reports, 9(1), 1–13. https://doi.org/10.1038/s41598-019-44892-y

This similarity takes a value between [0,1]. A larger value indicates that nodes with the same group membership tend to be close each other. Zero value means that membership *group_ids* is independent of the embedding.

The Element Centric Clustering Similarity is calculated as follows. 1. Construct a k-nearest neighbor graph. 2. For each edge connecting nodes i and j (i<j), find the groups g_i and g_j to which the nodes belong. 4. Make a list, L, of g_i's for nodes at the one end of the edges. Then, make another list, L', of nodes

at the other end of the edges.

5. Calculate the difference between L and L' using the Element Centric Clustering Similarity.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **group_ids** (*numpy.ndarray (num_entities,)*) – group membership for entities
- **A** (*scipy.csr_matrix, optional*) – precomputed adjacency matrix of the graph. If None, a k-nearest neighbor graph will be constructed, defaults to None
- **k** (*int, optional*) – Number of the nearest neighbors, defaults to 10

Returns element centric similarity

Return type float

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20)
>>> g = np.random.choice(10, 100)
>>> rho = emlens.element_sim(emb, g)
```

emlens.metrics.**f1_score**(*emb*, *target*, *agg='mode'*, ***params*)

Measuring the prediction performance based on the K-Nearest Neighbor Graph. Equivalent to knn_pred_score(emb, target, target_type = "disc").

This function measures how well the embedding space can predict the metadata of entities using the k-nearest neighbor algorithm. To this end, the following K-folds cross-validation is performed: 0. Split all entities into K groups. 1. Take one group as a test set and the other groups as a training set 2. Using the training set, predict the *target* variable for the entities in the training set. The prediction is made by the most frequent target variables of the nearest neighbors. 3. Calculate the prediction accuracy by the f1-score 4. Repeat Steps 1-3 such that each group is used as the test set once. 5. Compute the average of the prediction accuracy computed in Step 3.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **target** (*numpy.ndarray (num_target,)*) – target variable to predict
- **k** (*int or list, optional*) – Number of nearest neighbors, defaults to 10
- **n_splits** (*int, optional*) – Number of folds for the cross-validation, defaults to 10
- **iteration** (*int*) – Number of rounds of the cross validation. If iteration>1, the average of the cross validation score will be returned. If *return_score_all=True*, all scores will be returned, defaults to 1.
- **return_all_scores** (*bool*) – Set *True* to return all scores for the cross-vaidations. If set *False*, the mean of the score is returned
- **gpu_id** (*string or int*) – ID of the GPU device.
- **knn_exact** (*string or int*) – Set *True* to use the exact nearest neighbors for prediction. If set *False*, hueristics are used to find "probably" the nearest neighbors for the sake of substantial computation speed up.

Paramm metric Distance metric for finding nearest neighbors. Available metric *metric="euclidean"*, *metric="cosine"* , *metric="dotsim"*

Paramm agg How to aggregate the neighbors' variables. Setting *aggregation='mode'* uses the most frequent label, *='mean'* uses the mean as the predicted variable.

Returns dict object {"k", "score"}, where k is the number of neighbors, and score is the prediction score.

Return type dict

emlens.metrics.**find_mutual_edges**(*r*, *c*, *v=None*)

emlens.metrics.**find_nearest_neighbors**(*target*, *emb*, *k=5*, *metric='euclidean'*, *gpu_id=None*, *exact=True*)

Find the nearest neighbors for each point.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – vectors for the points for which we find the nearest neighbors

- **emb** – vectors for the points from which we find the nearest neighbors.
- **k** (*int, optional*) – Number of nearest neighbors, defaults to 5

Paramm metric Distance metric for finding nearest neighbors. Available metric *metric="euclidean"*, *metric="cosine"* , *metric="dotsim"*

Returns IDs of emb (indices), and similarity (distances)

Return type indices (numpy.ndarray), distances (numpy.ndarray)

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20)
>>> target = np.random.randn(10, 20)
>>> A = emlens.find_nearest_neighbors(target, emb, k = 10)
```

emlens.metrics.**knn_pred_score**(*emb*, *target*, *scoring_func*, *metric='euclidean'*, *agg='mode'*, *k=10*, *n_splits=10*, *iteration=1*, *return_all_scores=False*, *gpu_id=None*, *knn_exact=True*)

Prediction based on k-Nearest neighbor graph.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **target** (*numpy.ndarray (num_target,)*) – target variable to predict
- **scoring_func** (*numpy func*) – scoring function. This function will take a target variable *y* as the first argumebt and predicted variable *ypred* as the second argumebt, and ouputs the prediction score *score*, i.e., score=scoring_func(y, ypred).
- **k** (*int or list, optional*) – Number of nearest neighbors, defaults to 10
- **n_splits** (*int, optional*) – Number of folds, defaults to 10
- **iteration** (*int*) – Number of rounds of the cross validation. If iteration>1, the average of the cross validation score will be returned., defaults to 1.
- **return_all_scores** (*bool*) – Set *True* to return all scores for the cross-vaidations. If set *False*, the mean of the score is returned
- **gpu_id** (*string or int*) – ID of the GPU device.
- **knn_exact** (*string or int*) – Set *True* to use the exact nearest neighbors for prediction. If set *False*, hueristics are used to find "probably" the nearest neighbors for the sake of substantial computation speed up.

Paramm metric Distance metric for finding nearest neighbors. Available metric *metric="euclidean"*, *metric="cosine"* , *metric="dotsim"*

Paramm agg How to aggregate the neighbors' variables. Setting *aggregation='mode'* uses the most frequent label, *='mean'* uses the mean as the predicted variable. If there are more than k neighbors, aggregate the k neighbors connected by the edges with the largest weights, defaults to 'mode'

Returns dict object {"k", "score"}, where k is the number of neighbors, and score is the prediction score.

Return type dict

emlens.metrics.**linear_pred_score**(*emb*, *target*, *n_splits=10*, *iteration=1*, *return_all_scores=False*)

Measuring the prediction performance based on a linear regression model.

This function measures how well the embedding space can predict the metadata of entities using the linear regression model. To this end, the following K-folds cross-validation is performed: 0. Split all entities into K groups. 1. Take one group as a test set and the other groups as a training set 2. Using the training set, predict the *target* variable for the entities in the training set using a linear regression model. 3. Calculate the prediction accuracy 4. Repeat Steps 1-3 such that each group is used as the test set once. 5. Compute the average of the prediction accuracy computed in Step 3.

The performance score is measured based on the R^2 score.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **target** (*numpy.ndarray (num_target,)*) – target variable to predict
- **n_splits** (*int, optional*) – Number of folds, defaults to 10
- **iteration** (*int*) – Number of rounds of the cross validation. If iteration>1, the average of the cross validation score will be returned., defaults to 1.
- **return_all_scores** (*bool*) – "return_all_scores=True" or "=False" to return all scores in the cross validations or not, respectively.

Returns performance score

Return type float

emlens.metrics.**make_knn_graph**(*emb*, *k=5*, *binarize=True*, *metric='euclidean'*, *mutual=True*, *gpu_id=None*)

Construct the k-nearest neighbor graph from the embedding vectors.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **k** (*int or iterable, optional*) – Number of nearest neighbors. If list or array is given, then construct a k-nearest neighbor graph for each k, defaults to 5
- **binarize** – *binarize=False* will set the weight of the between nodes i and j by exp(-d_{ij]}). *binarize=True* will set to one., defaults to True

Paramm metric Distance metric for finding nearest neighbors. Available metric *metric="euclidean"*, *metric="cosine"* , *metric="dotsim"*

Returns The adjacency matrix of the k-nearest neighbor graph

Return type sparse.csr_matrix

```
>>> import emlens
>>> emb = np.random.randn(100, 20)
>>> A = emlens.make_knn_graph(emb, k = 10)
```

emlens.metrics.**modularity**(*emb*, *group_ids*, *k=10*, *metric='euclidean'*, *gpu_id=None*)

Calculate the modularity of entities with group membership. The modularity ranges between [-1,1], where a positive modularity indicates that nodes with the same group membership tend to be close each other. Zero modularity means that membership *group_ids* is independent of the embedding.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **group_ids** (*numpy.ndarray (num_entities,)*) – group membership for entities
- **A** (*scipy.csr_matrix, optional*) – precomputed adjacency matrix of the graph. If None, a k-nearest neighbor graph will be constructed, defaults to None

- **k** (*int, optional*) – Number of the nearest neighbors, defaults to 10

Paramm metric Distance metric for finding nearest neighbors. Available metric *metric="euclidean"*, *metric="cosine"* , *metric="dotsim"*

Returns modularity

Return type float

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20)
>>> g = np.random.choice(10, 100)
>>> rho = emlens.modularity(emb, g)
```

emlens.metrics.**nmi**(*emb*, *group_ids*, *A=None*, *k=10*, *metric='euclidean'*, *gpu_id=None*)

Calculate the Normalized Mutual Information for the entities with group membership. The NMI stands for the Normalized Mutual Information and takes a value between [0,1]. A larger NMI indicates that nodes with the same group membership tend to be close each other. Zero NMI means that membership *group_ids* is independent of the embedding.

NMI is calculated as follows. 1. Construct a k-nearest neighbor graph. 2. Calculate the joint distribution of the group memberships of nodes connected by edges 3. Calculate the normalized mutual information for the joint distribution.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **group_ids** (*numpy.ndarray (num_entities,)*) – group membership for entities
- **A** (*scipy.csr_matrix, optional*) – precomputed adjacency matrix of the graph. If None, a k-nearest neighbor graph will be constructed, defaults to None
- **k** (*int, optional*) – Number of the nearest neighbors, defaults to 10

Paramm metric Distance metric for finding nearest neighbors. Available metric *metric="euclidean"*, *metric="cosine"* , *metric="dotsim"*

Returns normalized mutual information

Return type float

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20)
>>> g = np.random.choice(10, 100)
>>> rho = emlens.nmi(emb, g)
```

emlens.metrics.**pairwise_distance**(*emb*, *group_ids*)

Pairwise distance between the centroid of groups. The centroid of a group is the average embedding vectors of the entities in the group.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding
- **group_ids** (*numpy.ndarray (num_entities)*) – group membership

Returns D, groups

Return type

numpy.ndarray, numpy.ndarray

- **D** (numpy.ndarray (num_groups, num_groups)): Distance matrix for groups.
- **groups** (numpy.ndarray (num_groups])): group[i] is the group for the ith row/column of D.

```
>>> import emlens
>>> import numpy as np
>>> import seaborn as sns
>>> emb = np.random.randn(100, 20)
>>> group_ids = np.random.choice(10, 100)
>>> D, groups = emlens.pairwise_distance(emb, group_ids)
>>> sns.heatmap(pd.DataFrame(D, index = groups, columns = groups))
```

emlens.metrics.**pairwise_dot_sim**(*emb*, *group_ids*)

Pairwise distance between groups. The dot similarity between two groups i and j is calculated by averaging the dot similarity of entities in group i and those in group j.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **group_ids** (*numpy.ndarray (num_entities)*) – group membership

Returns S, groups

Return type

numpy.ndarray, numpy.ndarray

- **S** (numpy.ndarray (num_groups, num_groups)): Similarity matrix for groups.
- **groups** (numpy.ndarray (num_groups])): group[i] is the group for the ith row/column of S.

```
>>> import emlens
>>> import numpy as np
>>> import seaborn as sns
>>> emb = np.random.randn(100, 20)
>>> group_ids = np.random.choice(10, 100)
>>> S, groups = emlens.pairwise_dot_sim(emb, group_ids)
>>> sns.heatmap(pd.DataFrame(S, index = groups, columns = groups))
```

emlens.metrics.**r2_score**(*emb*, *target*, *model='linear'*, *test=True*, ***params*)

Measuring the prediction performance based on the K-Nearest Neighbor Graph or Linear Regression.

If model == "knn", this is quivalent to knn_pred_score(emb, target, target_type = "cont").

This function measures how well the embedding space can predict the metadata of entities using the k-nearest neighbor algorithm. To this end, the following K-folds cross-validation is performed: 0. Split all entities into K groups. 1. Take one group as a test set and the other groups as a training set 2. Using the training set, predict the *target* variable for the entities in the training set. The prediction is made by the average target variables of the nearest neighbors. 3. Calculate the prediction accuracy by the R^2 score 4. Repeat Steps 1-3 such that each group is used as the test set once. 5. Compute the average of the prediction accuracy computed in Step 3.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors
- **target** (*numpy.ndarray (num_target,)*) – target variable to predict

- **model** – model to predict node attributes. With model="linear", the prediction is based on a linear regression model that predicts targets from the given embedding. With model="knn"i, the prediction is based on k-nearest neighbor graphs.

:type model:str :return: performance score :rtype: float

emlens.metrics.**rog**(*emb*, *metric='euc'*, *center=None*)

Calculate the radius of gyration (ROG) for the embedding vectors. The ROG is a standard deviation of distance of points from a center point. See https://en.wikipedia.org/wiki/Radius_of_gyration.

Parameters

- **emb** (*numpy.ndarray*) – embedding vector (num_entities, dim)
- **metric** (*str*) – The metric for the distance between points. The available metrics are cosine ('cos') euclidean ('euc') distances.
- **center** (*numpy.ndarray (num_entities, 1)*) – The embedding vector for the center location. If None, the centroid of the given embedding vectors is used as the center., defaults to None

Returns ROG value

Return type float

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20)
>>> rog = emlens.rog(emb, 'cos')
```

1.1.4 emlens.semaxis module

class emlens.semaxis.**LDASemAxis**(***params*)

Bases: *emlens.semaxis.SemAxis*

SemAxis based on Linear Discriminant Analysis.

A variant of SemAxis that finds the axis based on Linear Discriminant Analysis (LDA). This LDA-based SemAxis separates the given two groups more than the original SemAxis approach. The LDA-based SemAxis can find a "space" that best separates the groups. See https://en.wikipedia.org/wiki/Linear_discriminant_analysis.

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20) # Embedding vectors to ground the SemAxis
>>> group_ids = np.random.choice(2, 100) # Membership of entities
>>> target = np.random.randn(10, 20) # Vectors we will project onto the SemAxis
>>> model = emlens.LDASemAxis() # load SemAxis Object
>>> model.fit(emb, group_ids) # Fit the SemAxis
>>> model.transform(target, dim = 1) # Project `target` to the axis
>>> model.transform(target, dim = 2) # Project `target` to a 2D space
>>> model.save("random-semaxis.sm") # Save fitted SemAxis object
>>> model = emlens.LDASemAxis().load("random-semaxis.sm") # Load fitted SemAxis␣
↪object
```

transform(*target*, *dim=1*, ***params*)

Project the target vectors onto SemAxis.

Parameters **dim** (*int, optional*) – dimension for the projected space, defaults to 1

Returns Projected embedding vectors.

Return type numpy.ndarray (num_data,dim)

class emlens.semaxis.**SemAxis**

Bases: object

SemAxis Class object.

SemAxis aims to find an interpretable axis in the emebeding spacing using acronym entity groups. The axis is placed such that it runs through the centeroid of two acronym entity groups, and then all entities are projected to the axis.

Reference:

- [1] An, J., Kwak, H., & Ahn, Y.-Y. (2018). SemAxis: A Lightweight Framework to Characterize Domain-Specific Word Semantics Beyond Sentiment. Proc. the 56th Annual Meeting of the Association for Computational Linguistics, 1, 2450–2461.

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20) # Embedding vectors to ground the SemAxis
>>> group_ids = np.random.choice(2, 100) # Group membership of entities
>>> target = np.random.randn(10, 20) # Vectors we will project onto the SemAxis
>>> model = emlens.SemAxis() # load SemAxis Object
>>> model.fit(emb, group_ids) # Fit the SemAxis
>>> model.transform(target) # Project `target` to the axis
>>> model.save("random-semaxis.sm")
```

fit(*emb*, *group_ids*, *group_order=None*)

Finding the SemAxis from embedding vectors.

Parameters

- **emb** (*numpy.ndarray (num_entities, dim)*) – embedding vectors for locating the SemAxis
- **group_ids** (*numpy.ndarray (num_entities, dim)*) – group_ids, defaults to None.
- **group_order** (*list, optional*) – The axis points from group_order[0] to group_order[1]

load(*filename*)

Load SemAxis file.

Parameters **filename** (*str*) – filename

```
>>> import emlens
>>> xy = emlens.SemAxis().load('semspace.sm')
```

save(*filename*)

Save the fitted axis.

Parameters **filename** (*str*) – name of file

```
>>> import emlens
>>> import numpy as np
>>> emb = np.random.randn(100, 20)
>>> group_ids = np.random.choice(2, 100)
>>> model = emlens.SemAxis().fit(emb, group_ids)
>>> model.save('semspace.sm')
```

`transform`(*target*)
Project the target vectors onto SemAxis.

Parameters `target` (*numpy.ndarray (num_target, dim)*) – target embedding vectors to project onto the SemAxis.

Returns Projected embedding vectors.

Return type numpy.ndarray (num_data,)

1.1.5 emlens.vis module

1.1.6 Module contents

CHAPTER

TWO

INSTALL

See project page

CHAPTER

THREE

EXAMPLE

- airport networks

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

e

INDEX