

CS 214: Artificial Intelligence Lab

AI Lab Endsem

Common Instructions:

- The parent folder consists of three folders for three questions with the Python file and dataset required. (rename the parent folder as **rollno_AIlabendsem**)
- Open the Python files using IDLE and save the files in the folder you opened the file from.
- Save the zip file downloaded from moodle in `/home/user` The questions will effectively be in `/home/user/labendsem`
- Each question carries 10 marks. The total marks are 30.
- Fill in the statements given as underscores (_____) and remove the underscores when you fill in the code. Run your code and save the outputs in that same folder (the code for saving the plots is given in the Python file)
- Don't move out of specific folder for respective question. Evaluation will be done only if you save your code to the same folder as it is given.
- For each question, sufficient guidelines, and helper functions are provided. You may wish to use the numpy documentation given at the end of this paper. Try to utilize those to write your code.
- **Question 1:**
 - The treat will be at a fixed place on the maze. You will write code for Pacman to navigate through the corridors to the treat.
 - Implement the Uniform Cost Search (UCS) algorithm in the **uniformCostSearch** function in **search.py**.

You may observe *util.py* for any data structures that will be useful for your implementation. In addition to this, you will have to make use of the SearchProblem class in *search.py*

It contains methods such as *getStartState*, *isGoalState*, *getSuccessors* and *getCostofActions*. Autograder should pass all test cases for q3. The following command may be used to check if your solution is valid.

```
python3 autograder.py -q q3
```

Steps :

- Initialise priority queue and visited set
- Put a tuple containing start state and empty list of actions to the priority queue. Priority to be decided by cumulative path cost
- While the priority queue is not empty run the following
 - * Pop the Priority queue and set the current actions and current state to state returned by queue
 - * if current state is a goal state then return the current set of actions

- * if current state is not in visited Then
- * Add the current state to visited
- * Find successors of current state using getSuccessor
- * For every successor you have obtained
 - Append the state to the current actions set to construct the path
 - Calculate cost of path.
 - Push a tuple containing the path and the successor state to the priority queue with priority as path cost calculated in previous step
- * The while loop ends here
- Return an empty list if the while loop does not return a path.

• **Question 2:**

- Given a dataset consisting of X (bias included in the feature vector) and outputs Y :
- * Minimize :

$$\frac{1}{N} \sum_{i=1}^N (y_i - (\mathbf{x}_i^T \mathbf{w} + b))^2 + \lambda \|\mathbf{w}\|_2^2 \quad (1)$$

where \mathbf{x}_i corresponds to the i th row of the matrix X and y_i corresponds to the i th row of vector Y .

- * By constructing a ridge regression-based solution mapping the X and Y terms using the closed-form solution

$$w = (X^T X + \lambda I)^{-1} (X^T Y) \quad (2)$$

you can consider the regularization parameter $\lambda = 0.01$. Implement the closed-form solution.

- * Next, construct a ridge regression-based solution using the gradient descent update rule:

$$w = w - \eta \nabla w \quad (3)$$

$$\nabla w = -\left(\frac{2}{N}\right) X^T (y - Xw) + 2\lambda w$$

Where w is the weight vector, and η is the learning rate (you can consider $\lambda = 0.01$ or can pick some value and check for convergence). Implement the gradient update rule, run 1000 iterations of the gradient descent update rule, and report the final weight vector learnt.

- * Create two plots. In the first plot, depict the line learnt from the closed-form solution along with the given data, and in the other plot, depict the line learnt from the the gradient descent-based solution along with the given data. Comment your observation on the two learned solutions.

• **Question 3:** Implement the k-means algorithm to cluster the ‘data’ given as X , Fill in the function for k-means in the Python file, call the function, and, using the returned values, generate the plots. (Follow the steps given below to fill in the code)

- Randomly initialize K centroids at random locations within the data range.
- Store initial centroids for plotting
- Repeat till there is no change in the cluster centroids
 - * Calculate the Euclidean distance of each data point from the centroids.
 - * Assign the data points to the nearest cluster.
 - * Update the cluster centers
- At the end, plot the initial centroids and final centroids to depict the final clusters.
- Generate scatter plots of the given data whose color will depend on the final labels given by the K-means algorithm.

Some library functions for reference:

`numpy.mean(a, axis=None, dtype=None, out=None, keepdims=<no value>, *, where=<no value>)` [\[source\]](#)

Compute the arithmetic mean along the specified axis.


Returns the average of the array elements. The average is taken over the flattened array by default, otherwise over the specified axis. `float64` intermediate and return values are used for integer inputs.

Parameters: `a` : *array_like*

Array containing numbers whose mean is desired. If *a* is not an array, a conversion is attempted.

`axis` : *None or int or tuple of ints, optional*

Axis or axes along which the means are computed. The default is to compute the mean of the flattened array.

 *New in version 1.7.0.*

If this is a tuple of ints, a mean is performed over multiple axes, instead of a single axis or all the axes as before.

`dtype` : *data-type, optional*

Type to use in computing the mean. For integer inputs, the default is `float64`; for floating point inputs, it is the same as the input dtype.

`out` : *ndarray, optional*

example

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])
```

numpy.argmax

`numpy.argmax(a, axis=None, out=None, *, keepdims=<no value>)`

[\[source\]](#)

Returns the indices of the minimum values along an axis.

Parameters: **a** : *array_like*

Input array.

axis : *int, optional*


By default, the index is into the flattened array, otherwise along the specified axis.

out : *array, optional*

If provided, the result will be inserted into this array. It should be of the appropriate shape and dtype.

keepdims : *bool, optional*

If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the array.

 **New in version 1.22.0.**

Returns: **index_array** : *ndarray of ints*

Array of indices into the array. It has the same shape as *a.shape* with the dimension along *axis* removed. If *keepdims* is set to True, then the size of *axis* will be 1 with the resulting array having same shape as *a.shape*.

example

```
>>> a = np.arange(6).reshape(2,3) + 10
>>> a
array([[10, 11, 12],
       [13, 14, 15]])
>>> np.argmax(a)
0
>>> np.argmax(a, axis=0)
array([0, 0, 0])
>>> np.argmax(a, axis=1)
array([0, 0])
```

numpy.random.rand

random.rand(*d0*, *d1*, ..., *dn*)

Random values in a given shape.

Note

This is a convenience function for users porting code from Matlab, and wraps **random_sample**. That function takes a tuple to specify the size of the output, which is consistent with other NumPy functions like **numpy.zeros** and **numpy.ones**.

Create an array of the given shape and populate it with random samples from a uniform distribution over **[0, 1)**.

Parameters: *d0*, *d1*, ..., *dn* : *int, optional*

The dimensions of the returned array, must be non-negative. If no argument is given a single Python float is returned.

Returns: out : *ndarray, shape (d0, d1, ..., dn)*

Random values.

example

```
>>> np.random.rand(3,2)
array([[ 0.14022471,  0.96360618], #random
       [ 0.37601032,  0.25528411], #random
       [ 0.49313049,  0.94909878]]) #random
```

numpy.linalg.norm

`linalg.norm(x, ord=None, axis=None, keepdims=False)`

[\[source\]](#)

Matrix or vector norm.

This function is able to return one of eight different matrix norms, or one of an infinite number of vector norms (described below), depending on the value of the `ord` parameter.

Parameters: `x` : *array_like*

Input array. If *axis* is None, *x* must be 1-D or 2-D, unless *ord* is None. If both *axis* and *ord* are None, the 2-norm of `x.ravel` will be returned.

`ord` : *{non-zero int, inf, -inf, 'fro', 'nuc'}, optional*

Order of the norm (see table under [Notes](#)). `inf` means numpy's `inf` object. The default is None.

`axis` : *{None, int, 2-tuple of ints}, optional.*

If *axis* is an integer, it specifies the axis of *x* along which to compute the vector norms. If *axis* is a 2-tuple, it specifies the axes that hold 2-D matrices, and the matrix norms of these matrices are computed. If *axis* is None then either a vector norm (when *x* is 1-D) or a matrix norm (when *x* is 2-D) is returned. The default is None.

example

```
from numpy import linalg as LA
m = np.arange(8).reshape(2,2,2)
LA.norm(m, axis=(1,2))
array([ 3.74165739, 11.22497216])
LA.norm(m[0, :, :]), LA.norm(m[1, :, :])
(3.7416573867739413, 11.224972160321824)
```

numpy.linalg.inv

`linalg.inv(a)`

[\[source\]](#)

Compute the (multiplicative) inverse of a matrix.

Given a square matrix a , return the matrix $ainv$ satisfying `dot(a, ainv) = dot(ainv, a) = eye(a.shape[0])`.

Parameters: $a : (... , M, M)$ *array_like*
Matrix to be inverted.

Returns: $ainv : (... , M, M)$ *ndarray or matrix*
(Multiplicative) inverse of the matrix a .

Raises: `LinAlgError`
If a is not square or inversion fails.

example

```
>>> from numpy.linalg import inv
>>> a = np.array([[1., 2.], [3., 4.]])
>>> ainv = inv(a)
>>> np.allclose(np.dot(a, ainv), np.eye(2))
True
>>> np.allclose(np.dot(ainv, a), np.eye(2))
True
```

numpy.identity

numpy.identity(*n*, *dtype=None*, *, *like=None*)

[\[source\]](#)

Return the identity array.

The identity array is a square array with ones on the main diagonal.

Parameters: *n* : *int*


Number of rows (and columns) in $n \times n$ output.

dtype : *data-type, optional*

Data-type of the output. Defaults to `float`.

like : *array_like, optional*

Reference object to allow the creation of arrays which are not NumPy arrays. If an array-like passed in as *like* supports the `__array_function__` protocol, the result will be defined by it. In this case, it ensures the creation of an array object compatible with that passed in via this argument.

 *New in version 1.20.0.*

Returns: *out* : *ndarray*

$n \times n$ array with its main diagonal set to one, and all other elements 0.

example

```
>>> np.identity(3)
array([[1.,  0.,  0.],
       [0.,  1.,  0.],
       [0.,  0.,  1.]])
```


numpy.linspace

`numpy.linspace(start, stop, num=50, endpoint=True, retstep=False, dtype=None, axis=0) #` [\[source\]](#)

Return evenly spaced numbers over a specified interval.

Returns *num* evenly spaced samples, calculated over the interval *[start, stop]*.

The endpoint of the interval can optionally be excluded.

Changed in version 1.16.0: Non-scalar *start* and *stop* are now supported.

Changed in version 1.20.0: Values are rounded towards `-inf` instead of `0` when an integer *dtype* is specified. The old behavior can still be obtained with `np.linspace(start, stop, num).astype(int)`

Parameters: *start* : *array_like*

The starting value of the sequence.

stop : *array_like*

The end value of the sequence, unless *endpoint* is set to False. In that case, the sequence consists of all but the last of *num* + 1 evenly spaced samples, so that *stop* is excluded. Note that the step size changes when *endpoint* is False.

num : *int, optional*

Number of samples to generate. Default is 50. Must be non-negative

example

```
>>> np.linspace(2.0, 3.0, num=5)
array([2. , 2.25, 2.5 , 2.75, 3.  ])
>>> np.linspace(2.0, 3.0, num=5, endpoint=False)
array([2. , 2.2, 2.4, 2.6, 2.8])
>>> np.linspace(2.0, 3.0, num=5, retstep=True)
(array([2. , 2.25, 2.5 , 2.75, 3.  ]), 0.25)
```

numpy.dot

numpy.dot(*a*, *b*, out=None)

Dot product of two arrays. Specifically,

- If both *a* and *b* are 1-D arrays, it is inner product of vectors (without complex conjugation).
- If both *a* and *b* are 2-D arrays, it is matrix multiplication, but using `matmul` or `a @ b` is preferred.
- If either *a* or *b* is 0-D (scalar), it is equivalent to `multiply` and using `numpy.multiply(a, b)` or `a * b` is preferred.
- If *a* is an N-D array and *b* is a 1-D array, it is a sum product over the last axis of *a* and *b*.
- If *a* is an N-D array and *b* is an M-D array (where $M \geq 2$), it is a sum product over the last axis of *a* and the second-to-last axis of *b*:

```
dot(a, b)[i,j,k,m] = sum(a[i,j,:] * b[k,:,m])
```

It uses an optimized BLAS library when possible (see `numpy.linalg`).

Parameters: *a* : *array_like*

First argument.

b : *array_like*

example

```
>>> a = [[1, 0], [0, 1]]
>>> b = [[4, 1], [2, 2]]
>>> np.dot(a, b)
array([[4, 1],
       [2, 2]])
```

pandas.DataFrame.iloc

property `DataFrame.iloc`

[\[source\]](#)

Purely integer-location based indexing for selection by position.

⚠️ *Deprecated since version 2.2.0:* Returning a tuple from a callable is deprecated.

`.iloc[]` is primarily integer position based (from `0` to `length-1` of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. `5`.
- A list or array of integers, e.g. `[4, 3, 0]`.
- A slice object with ints, e.g. `1:7`.
- A boolean array.
- A `callable` function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above). This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.
- A tuple of row and column indexes. The tuple elements consist of one of the above inputs, e.g. `(0, 1)`.

`.iloc` will raise `IndexError` if a requested indexer is out-of-bounds, except *slice* indexers which allow out-of-bounds indexing (this conforms with python/numpy *slice* semantics).

example

```
>>> type(df.iloc[0])
<class 'pandas.core.series.Series'>
>>> df.iloc[0]
a    1
b    2
c    3
d    4
Name: 0, dtype: int64
```

pandas.read_csv

```
pandas.read_csv(filepath_or_buffer, *, sep=_NoDefault.no_default,
delimiter=None, header='infer', names=_NoDefault.no_default,
index_col=None, usecols=None, dtype=None, engine=None, converters=None,
true_values=None, false_values=None, skipinitialspace=False,
skiprows=None, skipfooter=0, nrows=None, na_values=None,
keep_default_na=True, na_filter=True, verbose=_NoDefault.no_default,
skip_blank_lines=True, parse_dates=None,
infer_datetime_format=_NoDefault.no_default,
keep_date_col=_NoDefault.no_default, date_parser=_NoDefault.no_default,
date_format=None, dayfirst=False, cache_dates=True, iterator=False,
chunksize=None, compression='infer', thousands=None, decimal='.',
lineterminator=None, quotechar='"', quoting=0, doublequote=True,
escapechar=None, comment=None, encoding=None, encoding_errors='strict',
dialect=None, on_bad_lines='error',
delim_whitespace=_NoDefault.no_default, low_memory=True, memory_map=False,
float_precision=None, storage_options=None,
dtype_backend=_NoDefault.no_default)
```

[\[source\]](#)

Read a comma-separated values (csv) file into DataFrame.

Also supports optionally iterating or breaking of the file into chunks.

Additional help can be found in the online docs for [IO Tools](#).

Parameters:

filepath_or_buffer : *str, path object or file-like object*

Any valid string path is acceptable. The string could be a URL. Valid URL schemes include http, ftp, s3, gs, and file. For file URLs, a host is expected. A local file could be: <file://localhost/path/to/table.csv>.

example

```
>>> pd.read_csv('data.csv')
```