SID500407020_code

October 21, 2020

[ ]:
```python
## was in decomposition.py
import numpy as np
import operator as op


def var_covar_matrix(X, mean=None, axis=0):
    assert len(X.shape) == 2, 'must operate on a matrix of 2 dimensions'
    if axis == 1:  # calculate on transpose
        return var_covar_matrix(X.T, mean=mean)
    elif axis != 0:
        raise ValueError('axis must 0 or 1')
    # axis is now == 0

    if mean is None:
        mean = np.mean(X, axis=axis)
    diff = X - mean

    # sum of outer products for each vector divided by the number of vectors
    rv = (diff.T @ diff) / X.shape[0]
    return rv


class Transformation:
    """Abstract Base Class for transformation objects."""

    def fit(self, X, y=None):
        """fit the transformation to data X

        :param X: input data (first dimension should represent rows)
        :param y: optional - data labels
        """
        raise NotImplementedError('This is an abstract method')

    def transform(self, X, y=None):
        """transform X into the representation domain.

        Raises an exception if fit has not first been called.
```

```python
        :param X: input data (first dimension should represent rows)
        :param y: optional - data labels
        """
        raise NotImplementedError('This is an abstract method')

    def inverse_transform(self, W, y=None):
        """Transform the representation back into the data domain.

        :param X: input data (first dimension should represent rows)
        :param y: optional - data labels
        """
        raise NotImplementedError('This is an abstract method')


class IdentityTransformation(Transformation):
    """IdentityTransformation. A transformation that does nothing

    useful for testing purposes
    """


    def __init__(self):
        pass

    def fit(self, X, y=None):
        pass

    def transform(self, X, y=None):
        return X

    def inverse_transform(self, W):
        return W

    def __str__(self):
        return 'IdentityTransformation()'


class PCA(Transformation):

    def __init__(self, components=None, normalize=True):
        self.normalize = normalize
        # assumes normalized data (mean of 0 over all axes, sd of 1)
        self.k = components
        self.normalize = normalize
        self._metavalues = dict(
                variances = None
```

```
                )

    def _norm(self, X):
        return (X - self.means) / self.sds

    def _inv_norm(self, W):
        return (W * self.sds) + self.means

    def fit(self, X, y=None):

        if self.k is None:
            self.k = X.shape[1]
        if self.normalize:
            self.means = np.mean(X, axis=0)
            self.sds = np.std(X, axis=0)
        else:
            self.means = np.zeros(X.shape[1])
            self.sds = np.ones(X.shape[1])

        X = self._norm(X)

        cov = var_covar_matrix(X, mean=np.zeros(X.shape[1]))
        val, vec = np.linalg.eigh(cov)  # cov is symmetric, so eigh performs better

        # vecs columns are eigenvectors
        pairs = sorted(zip(val, vec.T), key=op.itemgetter(0), reverse=True)
        self._metavalues['variances'] = np.array(list(map(op.itemgetter(0), pairs)))
        self.components = np.array(list(map(op.itemgetter(1), pairs[:self.k]))).T
        self.inverse_transform_components = self.components.T

    def transform(self, X):
        return self._norm(X) @ self.components

    def inverse_transform(self, W):
        return self._inv_norm(W @ self.inverse_transform_components)

    def __str__(self):
        return f'PCA({self.k}, {self.normalize})'


class NMF:

    def __init__(self, components, stop_threshold=0.01, max_iter=200, initial_dictionary=None, image_shape=None):
```

```
        if initial_dictionary is not None:
            initial_dictionary = initial_dictionary.copy()
        self._metavalues = dict(
            name='L2 Norm NMF',
            training_loss=[],
            training_residue=[],
            components=components,
            stop_threshold=stop_threshold,
            max_iter=max_iter,
            initial_dictionary=initial_dictionary,
            image_shape=image_shape,
        )
        self._dictionary = None
        self._inverse_dictionary = None

    def fit(self, X: np.ndarray, initial_representation=None):
        """ Assumes first dimension of X represents rows of data """

        if self._metavalues['image_shape'] is None:
            # initialise default image shape if was not previously assigned
            self._metavalues['image_shape'] = X.shape[1:]
        else:
            # sanity checks
            assert X.shape[1:] == self._metavalues['image_shape'], ('input data␣
↪does '
                                                                   'not match␣
↪expected shape')

        # reshape the data to be vectors instead of images (if not already␣
↪reshaped)
        n: int = X.shape[0]
        p: int = np.product(X.shape[1:])
        k: int = self._metavalues['components']
        X: np.ndarray = NMF._reshape_forward(X)
        assert X.shape == (p, n)

        # n - number of input images
        # p - dimensionality of population space
        # k - number of components

        # X shape (p, n)
        # D shape (p, k)
        # R shape (k, n)

        # initialise the learning dictionary if not already initialised
        if self._metavalues['initial_dictionary'] is None:
            self._metavalues['initial_dictionary'] = np.random.rand(p, k)
```

```
        else:
            assert self._metavalues['initial_dictionary'].shape == (p, k)

        # initialize dictionary if not already done.
        if self._dictionary is None:
            self._dictionary = self._metavalues['initial_dictionary'].copy()

        # initialize representation
        if initial_representation is None:
            R: np.ndarray = np.random.rand(k, n)
        else:
            R: np.ndarray = initial_representation.copy()
            assert R.shape == (k, n)

        D: np.ndarray = self._dictionary  # alias for readability.

        # toggle optimizing between D and R
        # start with updating 'R'
        optim = 'R'

        # marker for different calls
        self._metavalues['training_loss'].append(None)
        self._metavalues['training_residue'].append(None)

        # fit the data
        for iteration in range(self._metavalues['max_iter'] * 2):  # *2 to␣
↪account for alternation
            # this section follows section 2.7 of the accompanied documentation␣
↪in
            # ../papers/Robust Nonnegative Matrix Factorization using L21 Norm␣
↪2011.pdf

            # only collect the loss after D has been updated
            if optim == 'D':
                diffs = X - D @ R
                loss = l2_norm(diffs)
                residue = np.linalg.norm(diffs)

                # keep these for later
                self._metavalues['training_loss'].append(loss)
                self._metavalues['training_residue'].append(residue)

                # computing if stopping condition is met
                if iteration >= 2:
                    previous_loss = self._metavalues['training_loss'][-1]
                    current_loss = self._metavalues['training_loss'][-2]
```

```
                    relative_improvement= - (previous_loss - current_loss) /␣
↪previous_loss
                    if relative_improvement < self.
↪_metavalues['stop_threshold']:
                        optim = 'stop'

            if optim == 'D':
                optim = 'R'  # toggle for next time
                D *= (X @ R.T) / (D @ R @ R.T)
            elif optim == 'R':
                optim = 'D'
                R *= (D.T @ X) / (D.T @ D @ R)

            elif optim == 'stop':
                break
            else:
                assert 0, 'optim not recognised'

        self._inverse_dictionary = np.linalg.inv(D.T @ D) @ D.T

    def transform(self, X):
        """ Transform X into its representation

        :param X: row matrix/tensor of same shape as training time representing
            data. If there are n images of size 10x5, X should be of shape
            (n, 10, 5) or (n, 50) (depending on what was passed at training
            time)
        :return: row matrix (n, k) representing X.

        Returns a row oriented matrix of representation vectors of X
        """
        return (self._inverse_dictionary @ NMF._reshape_forward(X)).T

    def inverse_transform(self, R):
        """ Transform representations of X back into X.

        :param R: row oriented matrix of representation vectors
        :return: row matrix/tensor of the same shape as input (barring first
        dimension)
        """
        return self._reshape_backward(self._dictionary @ R.T)

    def get_metavalues(self):
        """ NMF.get_metavalues

        returns a dict with the following attributes:
        'name' : name of the algorithm
```

```
        'training_loss' : loss of the algorithm at each iteration during
            training
        'components' : dimensionality of the representation vectors.
        'max_iter' : maximum number of iterations in training
        """
        return self._metavalues

    @staticmethod
    def _reshape_forward(mat):
        """ transpose a row matrix or tensor to a column matrix """
        if len(mat.shape) == 3:
            return mat.reshape(mat.shape[0], -1).T
        if len(mat.shape) == 2:
            return mat.T
        raise ValueError(f'expected a 2 or 3 dimensional matrix. Got a matrix '
                         'of shape {mat.shape}')

    def _reshape_backward(self, mat):
        """transpose a column matrix to a row matrix / tensor (dependant on
        input to this class on training)"""
        assert len(mat.shape) == 2, 'needs a matrix not a tensor'
        newshape = self._metavalues['image_shape']
        return mat.T.reshape(mat.shape[1], *newshape)

    def __str__(self):
        D = self._metavalues
        return f"NMF({D['components']})"



def l2_norm(arr):
    return np.linalg.norm(arr)



## was in models.py
import numpy as np
from itertools import count

class TrivialModel:

    def __init__(self, rv=0):
        self.rv = rv

    def fit(self, x, y):
        pass
```

```python
    def predict(self, x):
        return [self.rv]*len(x)

    def __str__(self):
        return f'TrivialModel({self.rv})'


class GNB(TrivialModel):

    def __init__(self, sigma_adjust=1e-4):
        self.sigma_adjust = sigma_adjust

    def fit(self, x, y):
        self.levels = sorted(list(set(y)))
        self.means = []
        self.sds = []
        self.pclasses = []
        for i, level in enumerate(sorted(self.levels)):
            xcat = x[y==level]
            self.means.append(np.mean(xcat, axis=0))
            self.sds.append(np.std(xcat, axis=0))
            self.pclasses.append(len(xcat) / len(x))
        self.means = np.array(self.means)
        self.sds = np.array(self.sds)
        self.pclasses = np.array(self.pclasses)

    def norm_cdf(self, x, mu, sigma):
        """norm_cdf.

        :param x: ndarray (n by k) data
        :param mu: ndarray (k by c) means of each feature for each class
        :param sigma: ndarray (k by c) sd of each feature for each class
        :return: ndarray (n by k by c) probability contribution of each feature␣
↪being that value.
        """

        n, k = x.shape
        c = mu.shape[1]

        # sanity checks
        assert mu.shape[0] == k
        assert mu.shape == sigma.shape

        xt = np.transpose(np.tile(x, (c,1,1)), (1, 2, 0))
        mut = np.tile(mu, (n, 1, 1))
        sigmat = np.tile(sigma, (n, 1, 1)) + self.sigma_adjust
```

```
        # the above are now in the same shapes, meaning that *, +, -, /, **
        # all operate element-wise in a predictable way
        #>>> assert xt.shape == (n, k, c), f'{xt.shape} =/= {(n, k, c)}'
        #>>> assert mut.shape == (n, k, c), f'{mut.shape} =/= {(n, k, c)}'
        #>>> assert sigmat.shape == (n, k, c), f'{sigmat.shape} =/= {(n, k, c)}'

        inexp = -(xt - mut)**2 / (2 * sigmat ** 2)
        num = np.exp(inexp)
        den = np.sqrt(2 * np.pi * sigmat**2)

        rv =  num / den
        # sigmat can sometimes be 0 with homogenous features (features that
        # contribute nothing) This means that p(ci | xi) = 0 as xi xc for all␣
↪ci.
        return rv

    def predict(self, x):
        # tile and make of the form (n x k x c)
        ind_probs = self.norm_cdf(x, self.means.T, self.sds.T)
        conditionalprobs = np.sum(np.log(ind_probs), axis=1)

        # pclasses has shape (c,), so it is repeated for each row in␣
↪conditionalprobs * pclasses

        return np.argmax(conditionalprobs + np.log(self.pclasses), axis=1)

    def __str__(self):
        return f'GNB()'


def cosine_distance(X, Y):
    norms = np.linalg.norm(X, axis=1)[:, None] * np.linalg.norm(Y,␣
↪axis=1)[None, :]

    dot = np.zeros((X.shape[0], Y.shape[0]))
    for i, x in enumerate(X):
        for j, y in enumerate(Y):
            dot[i, j] = x @ y
    return 1 - dot / norms


class KNN:

    def __init__(self, k=3, distancefunction='euclidian', weigh_voting=True):
        self.k = k
        self.distfn = {
```

```
                'euclidian': lambda x, y: np.linalg.norm(x[:, None, :] -␣
↪y[None, :, :], axis=2),
                'manhatten': lambda x, y:np.sum(np.abs(x[:, None, :] - y[None, :
↪, :]), axis=2),
                'cosine': cosine_distance,
        }[distancefunction]
        self._name = f'{type(self).__name__}(k={k},␣
↪distancefunction={distancefunction})'
        self.weigh_voting = weigh_voting

    def fit(self, x, y):
        self.x = x
        self.y = y

    def predict(self, x, batchsize=10, k=None, dists=None, return_dists=False):
        if dists is None:
            all_dists = []
            for batch in breakup(x, batchsize):

                all_dists.append(self.distfn(batch, self.x))
            all_dists = np.concatenate(all_dists)
        else:
            all_dists = dists

        if k is None:
            k = self.k

        import datetime
        idxs = np.argpartition(all_dists, k)[:,:k]  # k smallest distance␣
↪indexes
        idxs_2 = np.argsort(all_dists, axis=1)[:,:k]
        if not self.weigh_voting:
            raise NotImplementedError()
            # TODO this method does not work - I must fix it
            res = np_mode(self.y[idxs], axis=(1,))
        else:
            all_dists[:, idxs]  # get k closest.
            # distances at those indexes
            distance_to = np.array([a[i] for a, i in zip(all_dists, idxs_2)])
            # true values at those indexes
            guesses = np.array([self.y[i] for i in idxs_2])
            # possible outcomes (may be a subset of range(10))
            possible = np.array(list(set(guesses.flatten())))
            # hardware; dimensions : represent:
            # 0 : data that is being predicted
            # 1 : possible labels the data could take
            # 2 : data that has known labels
```

```
            # actual values are True if that is the label for that known example
            mask = possible[None, :, None] == guesses[:, None, :]
            # mask the distances (at correct indecies) with the mask,␣
↪calculating contributions
            # then sum them
            distcats = np.sum(mask * 1/distance_to[:, None, :], axis=2)
            # calculate the indexes
            predidx = distcats.argmax(axis=1)
            # collect the indexes
            res = possible[predidx]

        if return_dists:
            return all_dists, res
        else:
            return res

    def __str__(self):
        return self._name


def np_mode(a, axis=None):
    """np_mode - numpy.mode implementation (it is not in base numpy).

    Works in the same way as np.mean, but returns the mode instead.

    :param ndarray: array to perform mode on
    :param axis: axis (or axes) over which to perform the mode.
    :return: ndarray of modes

    if a.shape == (w, x, y, z) and axis == (1, 2), then the returned value
    will have the shape (w, z).
    """
    if axis is not None:
        try:
            #axis = tuple(set(range(len(a.shape))) - set(axis))
            axis = tuple(axis)
        except TypeError:
            return np_mode(a, axis=(axis,))
    options = np.array([np.sum(a == i, axis=axis)
        for i in range(np.max((a)))
        ])
    return np.argmax(options, axis=0)


def breakup(itr, batchsize):
    """break up an iterable into more managable batches
```

```
    :param itr: iterable to break up
    :param batchsize: size of batch to break up into

    This will generate batches (of length batchsize) of iterable.
    """
    for batch in count():
        batch *= batchsize
        n = batch + batchsize
        if n < len(itr):
            yield itr[batch:n]
        else:
            break
    yield itr[batch:]


### was in model_tools.py

import random
import warnings
import numpy as np
from typing import List, Tuple
import h5py
import operator as op
import seaborn as sns
import traceback
from matplotlib import pyplot as plt


class Pipeline:

    def __init__(self, transformations):
        self._transformations = transformations

    def fit(self, X, Y, verbose=False):
        """ fit transformations to data and train the model with the output """
        for name, model in self._transformations[:-1]:
            if verbose:
                print(f'fitting transformation {name} (={model})')
            model.fit(X)
            X = model.transform(X)
        if verbose:
            print(f'fitting estimator {name} (={model})')

        self._transformations[-1][1].fit(X, Y)

    def run_transform(self, data, verbose=False):
        for name, model in self._transformations[:-1]:
```

```
            if verbose:
                print(f'transforming with {name} (={model})')
            data = model.transform(data)
        return data

    def predict(self, X, verbose=False, **pred_kwargs):
        return self._transformations[-1][1].predict(
                self.run_transform(X, verbose=verbose), **pred_kwargs)

    def __str__(self):
        return f'Pipeline({", ".join(map(op.itemgetter(0), self.
↪_transformations))})'


class CrossValidateClassification:
    """CrossValidateClassification.

    run cross-validation on a dataset with multiple models
    """

    def __init__(self, data: np.ndarray, labels: np.ndarray, n: int = 10,␣
↪verbose: bool = False):
        """__init__.

        if len(data) > len(labels) data is cut short to only include the first␣
↪len(labels) examples.

        :param data: input data (all data) MxN
        :type data: np.ndarray
        :param labels: input labels (all labels) N
        :type labels: np.ndarray
        :param n:
        :type n: int
        :return: [(true labels, predicted labels), ...]
        """

        idxs = list(range(len(labels)))
        random.shuffle(idxs)
        size = len(idxs)//n + 1
        validations = [set(idxs[i:i+size]) for i in range(0, len(idxs), size)]
        idxs = set(idxs)
        self.validation_groups = tuple(np.array(list(x)) for x in validations)
        self.data = data
        self.labels = labels
        self.verbose = bool(verbose)
```

```
    def run_validation(self, model: object) -> Tuple[List[np.ndarray], List[np.
↪ndarray]]:
        """run_validation.

        :param model_class: called to create the model. Must define fit and␣
↪predict methods.
        :type model_class: object
        :param args: arguments passed to model_class on instanciation.
        :param kwargs: keyword arguments passed to model_class on instanciation.
        :return: list of tuples of predicted observations and true observations
        :rtype: List[Tuple[np.ndarray, np.ndarray]]
        """
        res_true = []
        res_pred = []
        idxs = set(list(range(len(self.labels))))
        for i, idx_test in enumerate(self.validation_groups):
            if self.verbose:
                print(f'running fold #{i+1}      ')

            idx_train    = np.array(list(idxs - set(idx_test)))

            train_data   = self.data[idx_train]
            train_labels = self.labels[idx_train]

            test_data    = self.data[idx_test]
            test_labels  = self.labels[idx_test]

            model.fit(train_data, train_labels)
            pred = model.predict(test_data)
            res_true.append(test_labels)
            res_pred.append(pred)
        return res_true, res_pred

    @staticmethod
    def metrics(true, predicted=None, names=('accuracy',)):
        """metrics.

        :param true:
        :param predicted:
        """

        if predicted is None:  # attempt to unpack
            true, predicted = true

        cm = confusion_matrix(true, predicted)
        acc = np.mean(true == predicted)
        res = []
```

```
        if 'accuracy' in names:
            res.append(np.mean(true == predicted))
        return acc, cm  # prec, recall

    @staticmethod
    def aggregate_metrics(true, predicted):
        return np.array([CrossValidateClassification.metrics(t, p) for t, p in␣
↪zip(true, predicted)])

    def _random_idxs(self, n):
        idxs = list(range(len(self.labels)))
        random.shuffle(idxs)
        return idxs[:n]

    def random_sample(self, n):
        idxs = self._random_idxs(n)
        return self.data[idxs], self.labels[idxs]


def confusion_matrix(true, pred):
    rv = np.zeros([len(true)]*2)
    for t, p in zip(true, pred):
        rv[t, p] += 1
    return rv


def plot_confusion(confusion_matrix, title=None, labels=None, cmap='YlGnBu'):
    ax = sns.heatmap(confusion_matrix, linewidth=0.2, annot=True, cmap=cmap,␣
↪square=True)
    if labels is not None:
        ax.set_xticklabels(labels, rotation=90)
        ax.set_yticklabels(labels, rotation=0)
    if title is not None:
        ax.set_title(title)
    return ax


class ModelRunner:

    def __init__(self, *models):
        self.models = models

    def load_data(self, return_all=False):
        """ loads the datasets provided for this assignment """
        if hasattr(self, 'xtr'):
            return  # has already been run
```

```
        with h5py.File('./Input/train/images_training.h5','r') as H:
            self.xtr = np.copy(H['datatrain'])

        with h5py.File('./Input/train/labels_training.h5','r') as H:
            self.ytr = np.copy(H['labeltrain'])

        with h5py.File('./Input/test/labels_testing_2000.h5', 'r') as H:
            self.yte = np.copy(H['labeltest'])

        with h5py.File('./Input/test/images_testing.h5', 'r') as H:
            if return_all:
                self.xte = np.copy(H['datatest'])
            else:
                self.xte = np.copy(H['datatest'])[:len(self.yte)]

    def run_cv(self, folds=10, verbose=False):
        """ runs n-fold cross validation on the model """
        self.load_data()
        validator = CrossValidateClassification(self.xtr, self.ytr, n=folds,␣
↪verbose=verbose)
        results = {}
        for model in self.models:
            if verbose:
                print(f'running {model}')
            true, pred = validator.run_validation(model)
            results[str(model)] = CrossValidateClassification.
↪aggregate_metrics(true, pred)
        return results

    def run(self, n=None, verbose=False):
        """run all models and evaluate performance

        :param n: subset of test samples to evaluate model on.
        :param verbose: if true, print progress.
        """
        self.load_data()
        results = {}
        try:
            if n is None:
                n = len(self.xte)
        except ValueError:
            raise ValueError('could not interperate input')
        for model in self.models:
            if verbose:
                print(f'running {model}')
            try:
                model.fit(self.xtr, self.ytr)
```

```
            except Exception:
                tb = traceback.format_exc()
                warnings.warn(f'{tb}\nmodel {model} exited unexpectedly.'
                        '\nskipping...')
                continue  # skip this model
            results[str(model)] = CrossValidateClassification.metrics(
                    self.yte[:n], model.predict(self.xte[:n])
            )
            if verbose:
                print(f'{model} got {results[str(model)]}')
        return results


def confusion_matrix(true, pred, labels=None):
    if labels is None:
        labels = list(set(true))
    rv = np.zeros([len(labels)]*2)
    for t, p in zip(true, pred):
        rv[labels.index(p), labels.index(t)] += 1
    return rv


def plot_confusion_matrix(mat, labels):
    '''
    :param mat: confusion matrix
    :param labels: ordered labels to show on graph
    '''
    plt.imshow(mat)
    plt.xticks(range(len(labels)), labels, rotation='vertical')
    plt.yticks([-0.5] + list(range(len(labels))) + [len(labels) - .5],
               [''] + list(labels) + [''],
               rotation='horizontal')
```

[ ]:
```
best_model_predict = Pipeline([
    ('PCA(64, False)', PCA(64, normalize=False)),
    ('KNN(6, Manhatten)', KNN(6, 'manhatten', weigh_voting=True))
])
# using the whole dataset to train
mr = ModelRunner()
mr.load_data(True)
# concatenate
xtr, ytr, xte, yte, xpred = mr.xtr, mr.ytr, mr.xte[:len(mr.yte)], mr.yte, mr.
 ↪xte[len(mr.yte):]
best_model_predict.fit(np.concatenate((xtr, xte)), np.concatenate((ytr, yte)))
```

[ ]:
```
bm = Pipeline([
    ('PCA(64, False)', PCA(64, normalize=False)),
```

```
    ('KNN(6, Manhatten)', KNN(6, 'manhatten', weigh_voting=True))
])
bm.fit(xtr, ytr)
preds = bm.predict(xte)
```

[ ]:
```
plt.title(f'Confusion matrix of best selected model\nAccuracy={np.mean(preds ==␣
 ↪yte)}\n{bm}')
plot_confusion_matrix(confusion_matrix(yte, preds), labels=['T-shirt/
 ↪top','Trouser', 'Pullover', 'Dress', 'Coat', 'Sandal', 'Shirt', 'Sneaker',␣
 ↪'Bag', 'Ankle boot'])
```

[ ]:
```
predictions = best_model_predict.predict(xpred)
```

[ ]:
```
# save to h5 file
file = h5py.File('./Output/predicted_labels.h5', 'w')
file.create_dataset('predictions', data=predictions)
file.close()
```