CCE5225_Assignment_II_PGM_2020-21

January 30, 2021

1 Assignment - Probabilistic Graphical Models

1.0.1 Year 2020-2021- Semester I

1.0.2 CCE5225

1.1 #### Developed by - Adrian Muscat, 2020

Zachary Cauchi, 197999M, BSc CS, Yr I

Submit a pdf version (with the attached plagiarism form) of the final jupyter notebook (as a turn-it-in job on VLE) and the jupyter notebook itself separately (as an assignment job on VLE)

[1]:
```
import numpy as np
import pickle
import re

from skmultilearn.problem_transform import BinaryRelevance
from sklearn.utils._testing import ignore_warnings
from sklearn.exceptions import ConvergenceWarning
from sklearn.model_selection import GridSearchCV
from sklearn.preprocessing import LabelEncoder, OneHotEncoder,␣
↪MultiLabelBinarizer
from sklearn.linear_model import LogisticRegression
from sklearn.multioutput import ClassifierChain

import pandas
from collections import Counter

def saveAnswer(obj, name):
    answer_file = open(f'saved_answers/{name}.pkl', 'wb')
    pickle.dump(obj, answer_file)
    answer_file.close()

def trimSubClasses(labels):
    pattern = re.compile(r'.+?(?=_\d+(?!.))')
    # Ensure each object label does not contain a suffix of regex pattern "_\d"␣
↪(such as car_1, person_3, etc, returning car, person, etc instead)
    labels = [[label if not pattern.match(label) else pattern.match(label).
↪group(0) for label in row] for row in labels]
```

```
    return labels
```

[2]:
```
infile = open('MLC_data_2020_21.pkl','rb')
data = pickle.load(infile, encoding='latin1')
infile.close()
```

[3]:
```
train_obj_labels = trimSubClasses(data['development']['object_labels'])
train_out_labels = data['development']['output_labels']
train_geo_feat = data['development']['geometric_features']
test_obj_labels = trimSubClasses(data['test']['object_labels'])
test_out_labels = data['test']['output_labels']
test_geo_feat = data['test']['geometric_features']
```

2 Section 1: Preparing the data

2.1 Part 1

[4]:
```
# 1.a. Computing the mean output label count per example, per dataset␣
 ↪(development and test)
average_out_count_train = 0
average_out_count_test = 0

for row in train_out_labels:
    average_out_count_train += len(row)
for row in test_out_labels:
    average_out_count_test += len(row)

average_out_count_train /= len(train_out_labels)
average_out_count_test /= len(test_out_labels)

print('Answer to 1.a:')
print('Mean output labels per row (train set): ', average_out_count_train)
print ('Mean output labels per row (test set): ', average_out_count_test)

saveAnswer({
    'train_average_out': average_out_count_train,
    'test_average_out': average_out_count_test
}, '1a')
```

```
Answer to 1.a:
Mean output labels per row (train set):  2.1584763696214435
Mean output labels per row (test set):  2.148496240601504
```

[5]:
```
# 1.b. Flatten the output labels to a 1-d array, computing the distribution for␣
 ↪both datasets
```

```python
# Flatten the labels into a 1D array
flat_out_train = np.concatenate(train_out_labels)
flat_out_test = np.concatenate(test_out_labels)

# Count the numbers of each label
train_out_counts = Counter(flat_out_train)
test_out_counts = Counter(flat_out_test)

# Create dataframes from each counter object above.
train_out_counts_df = pandas.DataFrame.from_dict(train_out_counts,␣
 ↪orient='index')
train_out_counts_df.index.name = 'Label distribution in development (train) set'
test_out_counts_df = pandas.DataFrame.from_dict(test_out_counts, orient='index')
test_out_counts_df.index.name = 'Label distribution in test set'

print("Results for 1.b:")
display(train_out_counts_df)
display(test_out_counts_df)

saveAnswer({
    'train_out_counts': train_out_counts_df,
    'test_out_counts': test_out_counts_df
}, '1b')
```

```
Results for 1.b:

                                                 0
Label distribution in development (train) set
next_to                                       1411
at_the_level_of                                926
near                                          2276
behind                                        1055
opposite                                       267
on                                             359
in_front_of                                   1102
above                                          117
under                                          432
far from                                       376
against                                        593
outside_of                                      43
beyond                                          42
around                                          34
in                                              56
along                                           69
none                                            22

                                  0
Label distribution in test set
```

```
in_front_of                270
against                    136
next_to                    359
at_the_level_of            227
near                       578
under                      101
behind                     270
far from                   100
on                          88
opposite                    66
above                       31
along                       16
beyond                       5
around                       8
in                          18
outside_of                   8
none                         5
```

[6]:

```python
# 1.c. Computing the composite output labels (without flattening like in 1.b)␣
↪for both datasets.

# Same as above, compute the occurances of each composite output label.
# Unlike above, we first need to transform each row from an unhashable list to␣
↪a hashable tuple object.
train_cmp_out_counts = Counter(map(tuple, train_out_labels))
test_cmp_out_counts = Counter(map(tuple, test_out_labels))

train_cmp_out_counts_df = pandas.DataFrame.from_dict(train_cmp_out_counts,␣
↪orient='index')
train_cmp_out_counts_df.index.name = 'Composite output label distribution in␣
↪development (train) set'
test_cmp_out_counts_df = pandas.DataFrame.from_dict(test_cmp_out_counts,␣
↪orient='index')
test_cmp_out_counts_df.index.name = 'Composite output label distribution in␣
↪test set'

print('Results for 1.c:')
display(train_cmp_out_counts_df)
display(test_cmp_out_counts_df)

saveAnswer({
    'train_out_counts': train_cmp_out_counts_df,
    'test_out_counts': test_cmp_out_counts_df
}, '1c')
```

```
Results for 1.c:
```

0

```
Composite output label distribution in developm…
(next_to, at_the_level_of, near)                    509
(behind, opposite, near)                              3
(on,)                                               135
(in_front_of, near)                                 269
(near, behind)                                       31
…                                                 …
(opposite, beyond)                                    1
(in_front_of, opposite, under)                        1
(outside_of, next_to, at_the_level_of, near)          1
(in_front_of, next_to, opposite, near)                1
(against, at_the_level_of)                            1

[317 rows x 1 columns]

                                                    0
Composite output label distribution in test set
(in_front_of, against)                              7
(next_to, at_the_level_of, near)                  132
(under,)                                           53
(at_the_level_of,)                                  9
(in_front_of, next_to, at_the_level_of, near)       2
…                                                …
(above, next_to, against, behind, near)             1
(in, on)                                            1
(in_front_of, next_to, against)                     1
(around, against, near)                             1
(next_to, at_the_level_of, far from)                1

[166 rows x 1 columns]
```

[7]:

```python
# 1.d Computing the co occurrence probability distribution of the training set.

def computeProbabilityMatrix(matrix):
    # Get all the unique labels from the input matrix.
    labels = np.unique(np.concatenate(train_out_labels))

    # Create the output matrix for the probabilities.
    distribution = pandas.DataFrame(np.zeros(((len(labels)), (len(labels)))),␣
 ↪columns=labels, index=labels)
    distribution.index.name = 'co occurrence probabilities'

    # Get all co occurrences
    for sample in matrix:
        for target in labels:
            if target in sample:
                for prep in sample:
                    distribution[target][prep] += 1
```

```python
    targetCount = 0
    coCount = 0

    # Calculate the probabilities by dividing the 2nd labels co occurrences with the 1st labels total occurrences
    for target in labels:
        for prep in [label for label in labels if label != target]:
            targetCount = distribution[target][target]
            coCount = distribution[prep][target]

            distribution[prep][target] = coCount / targetCount

        # Set the targets to 0 to avoid bias
        distribution[target][target] = 0


    return distribution

probDistribution = computeProbabilityMatrix(train_out_labels)

display(probDistribution)

saveAnswer({
    'coOccurrenceProbMatrix': probDistribution
}, '1.d')
```

```
                             above   against     along    around  \
co occurrence probabilities
above                     0.000000  0.034188  0.000000  0.000000
against                   0.006745  0.000000  0.008432  0.008432
along                     0.000000  0.072464  0.000000  0.000000
around                    0.000000  0.147059  0.000000  0.000000
at_the_level_of           0.006479  0.109071  0.019438  0.000000
behind                    0.032227  0.067299  0.019905  0.001896
beyond                    0.047619  0.023810  0.000000  0.000000
far from                  0.026596  0.000000  0.002660  0.000000
in                        0.017857  0.232143  0.000000  0.000000
in_front_of               0.012704  0.060799  0.019964  0.000000
near                      0.029438  0.027241  0.023726  0.000000
next_to                   0.017009  0.096386  0.039688  0.000000
none                      0.000000  0.000000  0.000000  0.000000
on                        0.013928  0.571031  0.000000  0.000000
opposite                  0.022472  0.041199  0.000000  0.000000
outside_of                0.046512  0.069767  0.000000  0.000000
under                     0.000000  0.324074  0.002315  0.016204
```

co occurrence probabilities	at_the_level_of	behind	beyond	far from
above	0.051282	0.290598	0.017094	0.085470
against	0.170320	0.119730	0.001686	0.000000
along	0.260870	0.304348	0.000000	0.014493
around	0.000000	0.058824	0.000000	0.000000
at_the_level_of	0.000000	0.032397	0.001080	0.019438
behind	0.028436	0.000000	0.023697	0.152607
beyond	0.023810	0.595238	0.000000	0.452381
far from	0.047872	0.428191	0.050532	0.000000
in	0.000000	0.000000	0.000000	0.000000
in_front_of	0.036298	0.061706	0.009074	0.159710
near	0.315466	0.259227	0.002636	0.000439
next_to	0.532955	0.141035	0.000709	0.002126
none	0.000000	0.000000	0.000000	0.000000
on	0.000000	0.002786	0.000000	0.000000
opposite	0.172285	0.142322	0.011236	0.037453
outside_of	0.093023	0.302326	0.000000	0.232558
under	0.011574	0.087963	0.002315	0.011574

co occurrence probabilities	in	in_front_of	near	next_to	none
above	0.008547	0.119658	0.572650	0.205128	0.0
against	0.021922	0.112985	0.104553	0.229342	0.0
along	0.000000	0.318841	0.782609	0.811594	0.0
around	0.000000	0.000000	0.000000	0.000000	0.0
at_the_level_of	0.000000	0.043197	0.775378	0.812095	0.0
behind	0.000000	0.064455	0.559242	0.188626	0.0
beyond	0.000000	0.238095	0.142857	0.023810	0.0
far from	0.000000	0.468085	0.002660	0.007979	0.0
in	0.000000	0.000000	0.000000	0.000000	0.0
in_front_of	0.000000	0.000000	0.545372	0.189655	0.0
near	0.000000	0.264060	0.000000	0.512742	0.0
next_to	0.000000	0.148122	0.827073	0.000000	0.0
none	0.000000	0.000000	0.000000	0.000000	0.0
on	0.069638	0.050139	0.036212	0.005571	0.0
opposite	0.000000	0.269663	0.629213	0.179775	0.0
outside_of	0.000000	0.395349	0.441860	0.302326	0.0
under	0.000000	0.067130	0.164352	0.046296	0.0

co occurrence probabilities	on	opposite	outside_of	under
above	0.042735	0.051282	0.017094	0.000000
against	0.345700	0.018550	0.005059	0.236088
along	0.000000	0.000000	0.000000	0.014493
around	0.000000	0.000000	0.000000	0.205882
at_the_level_of	0.000000	0.049676	0.004320	0.005400
behind	0.000948	0.036019	0.012322	0.036019

beyond	0.000000	0.071429	0.000000	0.023810
far from	0.000000	0.026596	0.026596	0.013298
in	0.446429	0.000000	0.000000	0.000000
in_front_of	0.016334	0.065336	0.015426	0.026316
near	0.005712	0.073814	0.008348	0.031195
next_to	0.001417	0.034018	0.009213	0.014174
none	0.000000	0.000000	0.000000	0.000000
on	0.000000	0.000000	0.000000	0.000000
opposite	0.000000	0.000000	0.003745	0.018727
outside_of	0.000000	0.023256	0.000000	0.000000
under	0.000000	0.011574	0.000000	0.000000

2.2 Part 2

[8]:
```
#2.a Transform the object and geometrical features into an input matrix.

# Trim the file names from the inputs.
train_trimmed = np.array(train_obj_labels)[:, 1:]
test_trimmed = np.array(test_obj_labels)[:, 1:]

# Transform the features into one-hot encoded.
obj_encoder = OneHotEncoder(sparse=False)
obj_encoder = obj_encoder.fit(train_trimmed)
train_input_matrix = obj_encoder.transform(train_trimmed)
test_input_matrix = obj_encoder.transform(test_trimmed)

# Append the geometrical features onto the obtained one-hot features.
train_input_matrix = np.append(train_input_matrix, train_geo_feat, axis=1)
test_input_matrix = np.append(test_input_matrix, test_geo_feat, axis=1)

saveAnswer({
    'train_input_matrix': train_input_matrix,
    'test_input_matrix': test_input_matrix
}, '2.a')

XTrain = train_input_matrix
XTest = test_input_matrix
```

[9]:
```
# 2.b Transform the output features into a multi-label output matrix.

# Use a multi-label binarizer to one-hot encode and reduce multiple features␣
↪into a single vector.
out_one_hot = MultiLabelBinarizer()
out_one_hot = out_one_hot.fit(train_out_labels)

train_output_matrix = out_one_hot.transform(train_out_labels)
test_output_matrix = out_one_hot.transform(test_out_labels)
```

```python
saveAnswer({
    'train_output_matrix': train_output_matrix,
    'test_output_matrix': test_output_matrix
}, '2.b')

yTrain = train_output_matrix
yTest = test_output_matrix
```

2.3 Part 3

[10]:
```python
# 3 Functions for calculating accuracy metrics

def getMatrix(predictions, truths):
    # Generate a single confusion matrix for all labels
    tp = 0
    fp = 1
    fn = 2
    tn = 3

    # Initialise an empty array.
    matrix = [0, 0, 0, 0]

    # Over each prediction-truth pair, update the confusion matrix for that␣
↪label.
    for (plabel, tlabel) in np.nditer([predictions, truths], flags=['refs_ok']):
        if plabel == 1 and tlabel == 1: matrix[tp] += 1
        elif plabel == 1 and tlabel == 0: matrix[fp] += 1
        elif plabel == 0 and tlabel == 1: matrix[fn] += 1
        elif plabel == 0 and tlabel == 0: matrix[tn] += 1

    return matrix

def getMatrices(predictions, truths, num_labels):
    # Generate a multi-label confusion matrix
    tp = 0
    fp = 1
    fn = 2
    tn = 3

    # Initialise an empty set of arrays.
    matrices = [[0, 0, 0, 0] for i in range(0, num_labels)]

    it = np.nditer([predictions, truths], flags=['multi_index', 'refs_ok'])

    # Over each prediction-truth pair, update the confusion matrix for that␣
↪label.
```

```
    for plabel, tlabel in it:
        i = it.multi_index[1] # This is the label index
        if plabel == 1 and tlabel == 1: matrices[i][tp] += 1
        elif plabel == 1 and tlabel == 0: matrices[i][fp] += 1
        elif plabel == 0 and tlabel == 1: matrices[i][fn] += 1
        elif plabel == 0 and tlabel == 0: matrices[i][tn] += 1

    return matrices

# 3.a Accuracy (intersection over union)
def getAccuracy(predictions, truths):
    # Get the overall accuracy
    matrix = getMatrix(predictions, truths)

    correct = matrix[0] + matrix[3] # tp + tn
    total = sum(matrix) # tp + fp + fn + tn

    return correct / total

# 3.b Precision
def getPrecision(predictions, truths):
    # Get the overall precision
    matrix = getMatrix(predictions, truths)

    positives = matrix[0] # tp
    positiveGuesses = matrix[0] + matrix[1] # tp + fp

    return positives / positiveGuesses

# 3.c Recall
def getRecall(predictions, truths):
    # Get the overall recall
    matrix = getMatrix(predictions, truths)

    positives = matrix[0] # tp
    allPositives = matrix[0] + matrix[2] # tp + fn

    return positives / allPositives

# 3.d Per-label precision
def getMultiLabelPrecision(predictions, truths):
    # Get the per-label precision
    matrices = getMatrices(predictions, truths, len(truths[0]))

    precisions = [0 for i in range(len(matrices))]

    for i, (tp, fp, fn, tn) in enumerate(matrices):
```

```python
        p = tp + fp
        precisions[i] = (tp / p) if p != 0 else 0

    return precisions

# 3.e Per-label recall
def getMultiLabelRecall(predictions, truths):
    # Get the per-label recall
    matrices = getMatrices(predictions, truths, len(truths[0]))

    recalls = [0 for i in range(len(matrices))]

    for i, (tp, fp, fn, tn) in enumerate(matrices):
        allPositives = tp + fn
        recalls[i] = (tp / allPositives) if allPositives != 0 else 0

    return recalls
```

3 Section 2

3.1 Part 1

[11]:

```python
# 4.a Develop a binary-relevance model set using logistic regression, first␣
↪trained through cross-validation and then training the best br model on the␣
↪whole training set.

parameters = [
    {
        'classifier': [LogisticRegression()],
        'classifier__solver': ['sag', 'saga'],
        'classifier__C': [1.0, 0.5, 1.5],
        'classifier__max_iter': [250, 500, 100],
        'classifier__class_weight': [None, 'balanced'],
        'classifier__warm_start': [True],
        'classifier__random_state': [12]
    }
]

@ignore_warnings(category=ConvergenceWarning)
def getFittedBR(x, y):
    clf = GridSearchCV(BinaryRelevance(), parameters, scoring='accuracy',␣
↪verbose=2, n_jobs=4)
    return clf.fit(x, y)

brClf = getFittedBR(XTrain, yTrain)
display(brClf.best_params_, brClf.best_score_)
```

```python
print(f'Best estimator achieved an accuracy score of {brClf.best_score_} and␣
↪trained in {brClf.refit_time_:.2f} sec')

predictions = brClf.predict(XTest).todense()

saveAnswer({
    'trained_model': brClf,
    'predictions': predictions
}, '4.a')
```

```
Fitting 5 folds for each of 36 candidates, totalling 180 fits
```

```
[Parallel(n_jobs=4)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=4)]: Done  33 tasks      | elapsed:  1.1min
[Parallel(n_jobs=4)]: Done 154 tasks      | elapsed:  4.8min
[Parallel(n_jobs=4)]: Done 180 out of 180 | elapsed:  5.6min finished
```

```
{'classifier': LogisticRegression(max_iter=500, random_state=12, solver='sag',␣
↪warm_start=True),
 'classifier__C': 1.0,
 'classifier__class_weight': None,
 'classifier__max_iter': 500,
 'classifier__random_state': 12,
 'classifier__solver': 'sag',
 'classifier__warm_start': True}
```

```
0.0338582981958941
```

```
Best estimator achieved an accuracy score of 0.0338582981958941 and trained in
10.93 sec
```

[12]:

```python
# 4.b Metrics on the trained models from 4.a



# Next, compute the metrics accordingly.
brAcc = getAccuracy(predictions, yTest)
brPre = getPrecision(predictions, yTest)
brRec = getRecall(predictions, yTest)
brPrePerLabel = getMultiLabelPrecision(predictions, yTest)
brRecPerLabel = getMultiLabelRecall(predictions, yTest)

# In the case of the multi-label metrics, convert them into dataframes for␣
↪readability.
brPrePerLabel = pandas.DataFrame(brPrePerLabel, index=out_one_hot.classes_,␣
↪columns=['BR Precision'])
brRecPerLabel = pandas.DataFrame(brRecPerLabel, index=out_one_hot.classes_,␣
↪columns=['BR Recall'])

print(f'Accuracy: {brAcc}')
```

```python
print(f'Precision: {brPre}')
print(f'Recall: {brRec}')
display(brPrePerLabel)
display(brRecPerLabel)

saveAnswer({
    'accuracy': brAcc,
    'precision': brPre,
    'recall': brRec,
    'multiPrecision': brPrePerLabel,
    'recPerLabel': brRecPerLabel
}, '4.b')
```

```
Accuracy: 0.8857253427686864
Precision: 0.6102719033232629
Recall: 0.2650918635170604
```

	BR Precision
above	0.000000
against	0.000000
along	0.000000
around	0.000000
at_the_level_of	0.000000
behind	0.681818
beyond	0.000000
far from	0.625000
in	0.000000
in_front_of	0.672131
near	0.597605
next_to	0.000000
none	0.000000
on	0.000000
opposite	0.000000
outside_of	0.000000
under	1.000000

	BR Recall
above	0.000000
against	0.000000
along	0.000000
around	0.000000
at_the_level_of	0.000000
behind	0.222222
beyond	0.000000
far from	0.050000
in	0.000000
in_front_of	0.151852
near	0.863322

```
next_to          0.000000
none             0.000000
on               0.000000
opposite         0.000000
outside_of       0.000000
under            0.009901
```

3.2 Q5: Construct a bayesian network

Aim is to construct a bayesian network using the above-generated co occurrence probability distribution.

3.2.1 Associated labels

First would be to identify the strongest-pairing labels to other labels (and vice-versa to obtain dead-end labels/pairs).

Immediately, it can be concluded that every variable is independent of `none`.

Otherwise, getting the co occurrences with the a probability of at least 0.4, we get to following:

[13]:
```python
out_labels = np.unique(np.concatenate(train_out_labels))
top_matches = pandas.DataFrame(columns=['target', 'prep', 'P(target|prep)'])

# Get the probabilities matching our criteria.
for target in out_labels:
    for prep in out_labels:
        if probDistribution[prep][target] >= 0.4:
            top_matches.loc[len(top_matches)] = [target, prep,␣
↪probDistribution[prep][target]]

# Sort them in descending order by probability.
top_matches.sort_values(by=['P(target|prep)'], ascending=False,␣
↪ignore_index=True)
```

[13]:

	target	prep	P(target|prep)
0	next_to	near	0.827073
1	at_the_level_of	next_to	0.812095
2	along	next_to	0.811594
3	along	near	0.782609
4	at_the_level_of	near	0.775378
5	opposite	near	0.629213
6	beyond	behind	0.595238
7	above	near	0.572650
8	on	against	0.571031
9	behind	near	0.559242
10	in_front_of	near	0.545372
11	next_to	at_the_level_of	0.532955
12	near	next_to	0.512742

13	far from	in_front_of	0.468085
14	beyond	far from	0.452381
15	in	on	0.446429
16	outside_of	near	0.441860
17	far from	behind	0.428191

Based on the above, dependencies can be drawn from the pairs to form a basic bayesian network, modelling the probability that any one state will also include any other connected state.

There are also two cyclic links from the above, which need to be removed. Doing so leaves the following.

The graph is divided into two components, with the `near` label being the root of the greatest of the two graphs. It does not include the `around` or `under` labels, which did not have a very strong dependency on the graph nodes. `none` is also not shown as it is not dependent on any other label (since `none` cannot be derived/derive any other label).

This leaves the above graph, which can be converted into a chain order for training the bayesian chain classifier.

[14]:
```python
print("Output labels:")
display(out_labels)

# Order derived from bayesian network above
chain_order_1 = [10, 11, 2, 4, 14, 0, 5, 9, 7, 6, 15, 1, 13, 8, 12, 3, 16]
print(f'Derived chain order from above network is:\n{[out_labels[i] for i in␣
↪chain_order_1]}')
```

```
Output labels:

array(['above', 'against', 'along', 'around', 'at_the_level_of', 'behind',
       'beyond', 'far from', 'in', 'in_front_of', 'near', 'next_to',
       'none', 'on', 'opposite', 'outside_of', 'under'], dtype='<U15')

Derived chain order from above network is:
['near', 'next_to', 'along', 'at_the_level_of', 'opposite', 'above', 'behind',
'in_front_of', 'far from', 'beyond', 'outside_of', 'against', 'on', 'in',
'none', 'around', 'under']
```

[15]:
```python
# 6: Retrain the model using the bayesian network

@ignore_warnings(category=ConvergenceWarning)
def getFittedBCC(x, y, order):
    chainParams = {
        'order': [
            order,
        ]
    }

    clf = GridSearchCV(ClassifierChain(LogisticRegression()), chainParams)
```

```
    return clf.fit(x, y)

bccClf = getFittedBCC(XTrain, yTrain, chain_order_1)

bccPredictions = bccClf.predict(XTest)
```

[16]:
```
# Get the metrics
bccAcc = getAccuracy(bccPredictions, yTest)
bccPre = getPrecision(bccPredictions, yTest)
bccRec = getRecall(bccPredictions, yTest)
bccPrePerLabel = getMultiLabelPrecision(bccPredictions, yTest)
bccRecPerLabel = getMultiLabelRecall(bccPredictions, yTest)

# Convert them into dataframes
bccPrePerLabel = pandas.DataFrame(bccPrePerLabel, index=out_one_hot.classes_,␣
↪columns=['BCC Precision'])
bccRecPerLabel = pandas.DataFrame(bccRecPerLabel, index=out_one_hot.classes_,␣
↪columns=['BCC Recall'])

# Concatenate the final BCC metrics with the BR metrics for comparison.
finalPerLabelPre = pandas.concat([brPrePerLabel, bccPrePerLabel], axis=1)
finalPerLabelRec = pandas.concat([brRecPerLabel, bccRecPerLabel], axis=1)
finalPerLabelMetrics = pandas.concat([finalPerLabelPre, finalPerLabelRec],␣
↪axis=1)
finalMetrics = pandas.DataFrame([[brAcc, bccAcc], [brPre, bccPre], [brRec,␣
↪bccRec]], index=['Accuracy', 'Precision', 'Recall'], columns=['BR', 'BCC'])

print('\nFinal metrics:')
display(finalMetrics)
print('\nFinal per-label metrics:')
display(finalPerLabelMetrics)

saveAnswer({
    'bnChain': bccClf.best_params_['order'],
    'classifier': bccClf,
    'finalMetrics': finalMetrics,
    'finalPerLabelMetrics': finalPerLabelMetrics
}, '6')
```

Final metrics:

	BR	BCC
Accuracy	0.885725	0.888545
Precision	0.610272	0.560268
Recall	0.265092	0.548994

```
Final per-label metrics:
                 BR Precision  BCC Precision  BR Recall  BCC Recall
above                0.000000       0.600000   0.000000    0.096774
against              0.000000       0.389831   0.000000    0.338235
along                0.000000       1.000000   0.000000    0.125000
around               0.000000       1.000000   0.000000    0.125000
at_the_level_of      0.000000       0.432886   0.000000    0.568282
behind               0.681818       0.632124   0.222222    0.451852
beyond               0.000000       0.000000   0.000000    0.000000
far from             0.625000       0.518072   0.050000    0.430000
in                   0.000000       0.500000   0.000000    0.111111
in_front_of          0.672131       0.514894   0.151852    0.448148
near                 0.597605       0.692542   0.863322    0.787197
next_to              0.000000       0.511848   0.000000    0.601671
none                 0.000000       0.000000   0.000000    0.000000
on                   0.000000       0.522936   0.000000    0.647727
opposite             0.000000       0.200000   0.000000    0.015152
outside_of           0.000000       0.000000   0.000000    0.000000
under                1.000000       0.532710   0.009901    0.564356
```

4 Q7: Evaluation of results

The performance results for both models are quite interesting, in that both achieved a near-equal accuracy of 89%. BCC sees a decline in precision of 5% compared to BR, suggesting it returns fewer quality results compared to the BR model. Looking at the per-label distribution, however, shows that the BCC model in fact performs better on most labels. The BR model did not make any predictions on some labels, only predicting with a handful of labels.

BCC saw the largest improvement in terms of recall, with an overall score of 55% compared to the 27% of the BR model. Looking at the per-label recall shows that the BR model only predicted positives for a handful of labels. BCC seemed to make a greater variety of predictions, going for labels which the BR model didn't make any predictions on.

The above results suggest that the BR model tended towards only a handful of labels as possible predictions, likely due to it not correlating any possible relations between the labels. BCC does not seem to face this issue, being able to make multiple label predictions for a single sample and trying other labels as predictions. This is in large part to the order of labels in which it is trained as derived from the bayesian network modelled above. Based on these results, it appears as though the BCC definitely improved over the BR model based on the higher recall and precision, with a relatively smaller reduction in precision. This can be an acceptable tradeoff depending on the nature of the multi-label problem, the relationships between the labels presented and what metrics will be more favourable in that problem. For those which do not see any relation between many/all of the labels, it might be suitable to use the Binary Relavance model. For those problems which have multiple intercompatible labels (such as this scenario where `near` might frequently be classified alongside `next_to`), having a model such as the Bayesian Chain Classifier being aware of those relationships may show improved performance as it did here.