Graph Algorithms for Feature Extraction in Music Data

Sebastian Coates

May 12, 2018

Abstract

Music Information Retrieval (MIR) is a broad field with various goals such as genre classificaiton, artist classification, and music generation. A common challenge in MIR is how to represent and analyze music digitally. Common representations include digital audio formats, like .mp3 and .wav, and MIDI, which contains more precise note information but lacks information about timbre. Both formats include a large amount of information; machine learning tasks in MIR seek to reduce this information and extract important features. A common approach to condensing the information in audio formats is mel-frequency ceptstrum coefficiencs (MFCC). This paper presents a new alternative: a way of modeling MIDI information with a directed graph. This paper demonstrates that eight features extracted from this graph alone are able to achieve .98 accuracy in a binary genre classification task and .92 accuracy on a binary artist classification task. Furthermore, this paper discusses potential alterations and improvements to the model as well as potential applications to music generation.

Introduction

Overview

The aim of this research is to explore a novel, graph-based approach to music information retrieval (MIR). Songs are essentially a collection of notes, related by the order in which they're played. As such, they can be effectively modeled as a graph. In this project, songs were modeled as a digraph and features were extracted from the graph using various graph algorithms. These features were be used in machine learning applications, including classification of song by genre and composer.

Graph Representation

A weighted digraph $G = (V, E, \omega)$ is generated based on the contents of a single-instrument song S as follows:

$$\exists V_{i,j} \in V \iff \exists \text{ note of pitch i and duration j in } S.$$

$$\omega(E_{x,y}) := \text{number of times note y is played immediately following note x.}$$

Chords, or any simultaneously played notes, are represented as cliques.

To generalize to multi-instrument songs, a separate graph can be generated for each instrument's part. Another option is to merge all Midi tracks and treat simultaneous notes as chords.

For this paper, the latter option was chosen when a piano recording was separated the right and left hand. See **Figure 1** for an example graph.

Assumptions and Constraints

It cannot be assumed that Midi files provide a perfect representation of a song. Midi data does not contain much information about the timbre of a piece of music. Midi does contain 'velocity' information, often used as a measure of how hard a piano key was pressed, for example. This allows for encoding some aspect of timbre into Midi data. Additionally, Midi data does not represent a song exactly as sheet music would. While Midi can be used to represent exact note lengths, this is generally not the case. Midi data is often generated by recording human playing on a keyboard, producing imperfect note lengths. As such, it was necessary to quantize note lengths. As a result of this discritization, only 31 note lengths are possible.

The Midi parsing and graphical representation result in information loss. 'Swing', a rhymic quality of jazz music, was not able to be encoded, as it is ignored by the quantization process. Furthermore, musical rests were not incorporated in the definition of the graph nor the process of parsing Midi notes. This was done to limit the size and scope of the graph. However, it means that an important aspect of musical composition is not effectively encoded in the graph representation. For similar reasons and with similar consequences, note velocity was ignored.

Classification

In order to test the efficacy of this graph representation, its features were used in two binary classification problems: classifying the genre of a song (classical or jazz), and classifying the composer of a classical composition (Bach or Chopin). As an added constraint, all songs selected were piano-only.

The classical category includes 150 songs made up of compositions by the following composers: Beethoven, Brahms, Chopin, Grieg, Schubert, and Schumann [1]. The jazz category includes 150 songs, primarily jazz standards, written by varying composers and recorded by the same musician [2]. Each genre was divided up into 120 songs for training and 30 songs for testing.

For composer classification, 120 midi files were gathered for Bach and Chopin [3]. The same ratio of training-test data was used for this classifcation problem, resulting in 96 songs for training and 24 songs for testing for each composer.
For both classification problems, four machine learning models were used: k-nearest neighbors (KNN), decision tree, adaboosting, and gradient doosting tree classifiers were used [4]. KNN and decision tree classifiers are fairly simple, and provide clear insight into how features are utilized in the learning process. The latter ensemble methods were used to maximize classification performance and determine the overall efficacy of the graph and feature selection process.

Feature Selection

Eight features were selected from each graph for the classifaction problems: the number of nodes, the sum of the edge weights, the number of strongly connected components, the size of largest strongly connected component, the average distance from the first node to other nodes in a shortest path tree, the number of unmatched vertices, the average edge weight, and the total weight of self-loops. The features were chosen because they encode different aspects of the graph, and, ideally, these features are independent. However, it's likely that there exists correlation between all of the features.

Various algorithms are required to extract these features ranging from simple to complex. For a discussion of these algorithms, see **Analysis of Graph Algorithms**.

Related Works

Alberto Pinto et al. use a similar approach to modeling music using a graph[5]. Their model seeks to understand the relationship between notes as intervals in order to determine the similarity of melodies. They approach feature extraction from the perspective of graph spectra. Their paper shows that this spectra-based feature extraction is able to effectively indexing Dutch folk songs. The paper demonstrates that a graph representation of music and its corresponding features can be a very effective tool for machine learning applications.

Amit Tiroshi et al. approach a tangential problem: recommendation using graph-based features[6]. The general approach to their graph analysis draws from classic examples of random walks to build recommendations. It also discusses how graph feature extraction has been used to enhance existing models. An important take-away from this work is that the authors do not consider large graphs as a whole. Instead of analyzing the entire adjacency matrix or laplacian matrix of a graph, they demonstrate that features extracted by graph algorithms can provide an effective, condensed representation of the graph.

"A Survey of Audio-Based Music Classification and Annotation" discusses modern approaches to many categories of music classification, including genre [7]. In addition to discussing classification broadly, this paper speaks in depth about Mel-frequency cepstral coefficients (MFCCs). MFCCs are an effective tool for feature extraction of audio data. Much work is being done with MFCCs, as they are commonly applied to music classification problems. While MFCCs are not graph-based, they share the philosophy of extracting important information from music data.

Michael Haggblade, Yang Hong, and Kenny Kao explored the problem of music genre classification in CS229 at Stanford University[8]. They provide benchmarks for relevant models, including KNN, SVM, and neural networks. They used MFCC for feature extraction, which differs significantly from a graph representation as explained above. They compared four distinct genres: classical, jazz, metal, and pop and achieve accuracy of 80% overall. These results demonstrate the efficacy of MFCCs and provide an important point of comparison. The paper also provides benchmarks in the form of confusion matrices, which demonstrate the accuracy of their predictions. These confusion matrices provide specific insight into how MFCCs are able to distinguish between jazz and classical music.

Kenwoo Choi et al. provide an overview of deep learning for music information retrieval. Deep networks are able to effectively learn non-linear relationships between features. Music provides quite a complicated feature-space, whether or not MIR or a graph representation is used. This paper discusses how neural networks can be effective in learning these complicated relationships. [9].

Analysis of Graph Algorithms

Various graph algorithms are necessary for feature extraction on each graph, including advanced algorithms like Tarjan's Algorithm, which finds the connected components of the graph, Dijkstra's Algorithm, which finds the shortest path from the first vertex to all the other vertices, and a greedy maximal matching algorithm. Dijkstra's algorithm relies on every edge weight being non-negative, which is satisfied by the graph definition. Tarjan's algorithm is useful for directed graphs, so its appropriate in this context. Maximal matching is applicable to weighted graphs, as it tries to find the maximally weighted matchings as quickly as possible, so it is also useful for this problem.

These advanced algorithms are relatively computationally expensive, becuase they iterate over the vertex set and edge set in a graph. In practice, these algorithms were slow enough to make feature extraction non-trivial. The number of vertices in more complicated songs was quite large due to the graph definition. This makes the advanced algorithms particularly expensive. The features extracted by these algorithms, as discussed later, are quite effective, so their complexity is a worthwhile trade-off.

The simpler-to-extract features just required computation on the edge set of the graph. In practice, the graphs were sparse, making these features very quick to extract. While it's possible to compose a song that is represented by a dense graph, it seems that most songs are sparse in nature. This is a benefit of the graph definition, as it chooses to have weighted edges instead of having multi-edges.

On one computer, it took 15 times longer to extract the more complicated features than extract the simple features. This is in part due to utilization of efficient Numpy code in the simpler feature extraction, but the difference is still quite significant.

While feature extraction is an expensive process, it can be easily made parallel. Each feature is independent to compute, so each graph algorithm can be run in parallel to extract the necessary features. This would allow the feature extraction process to scale well to much larger datasets than those analyzed in this paper, provided that multi-core architectures are available.

Additionally, in the broader context of classification problems, feature extraction is not a significant burden, as it only has to be done once. During the process of cross-validation and hyperparameter selection, the features are extracted once at the beginning and can be reused during each stage of training.

The classification results provide deeper insight into the efficacy of each graph algorithm.

Results

Genre Classification

Each feature was tested individually in the genre classification problem to gain a sense of the efficacy of each feature. Ultimately, the features were combined in order to achieve as high a classification accuracy as possible. The results are depicted in the following table:

Features Used	Accuracy
Number of Nodes	.83
Sum of Weights	.72
Tarjan's	.77
Dijkstra's	.93
Matching	.48
Average Weight	.90
Self Loops	.70
All features	**.98**

Each feature, except for matching, seems to be quite effective on its own. This both reflects the significant difference in style between the genres and the efficacy of the graph at representing the music. The results of Dijkstra's algorithm are the most effective feature in distinguishing classical music from jazz. While it's hard to say exactly why this is the case, it's possible that the more complicated harmony or improvisation in jazz music effects the shortest path tree in a significant way.

In the results of Haggblade, Hong, and Kao, jazz appears to be the most difficult genre to classify, achieving the lowest accuracy in 3 of 4 classifiers. Furthermore, their models confuse jazz and classical music often; with their k-means classification, 16 songs are predicted to be jazz when they are actually classical. This is significant compared to the 27 songs that are correctly predicted to be jazz and the 14 songs that are correctly predicted to be classical [8]. While their MFCC approach to classification

struggled to compare jazz and classical music, this graph-based approach seems to have no such trouble. This suggests that a graph-based approach might be more effective than MFCC for certain classification problems.

When combining all the features, the overall accuracy is quite high, demonstrating that the graph representation and feature extraction are quite an effective tool for genre classification.

Composer Classifcation

Like with genre classification, each feature was tested individually to assess its significance. Lastly, the features were all combined to determine their combined efficacy. This was expected to be a more difficult classification problem than genre classification, as both artists compose music of the same genre.

Features Used	Accuracy
Number of Nodes	.77
Sum of Weights	.85
Tarjan's	.81
Dijkstra's	.54
Matching	.65
Average Weight	.52
Self Loops	.73
All features	**.92**

Most features seem effective on their own, except for average weight and Dijkstra's. Combining all the features produces a high accuracy, one that's significantly higher than the features individually. This demonstrates that while the features are effective on their own, they are much more effective in a broader context. Furthermore, this suggests that there is not significant redundnacy in the features; eliminating a feature from consideration does impact the results. The overall accuracy of this more difficult classification challenge reaffirms the efficacy of the feature selection process and graph representation.

Comparing the results of the two classification problems results in some important insight. The overall accuracy for the full feature set is comparable for both classification problems. That the artist classification problem results in a lower accuracy is to be expected. Jazz and classical are quite distinct genres while Bach and Chopin produce music of a much more similar style. Comparing the results of individual features provides interesting results as well. Notably, the results of Dijkstra's algorithm alone are able to differentiate jazz from classical music 90% of the time. However, in comparing Bach to Chopin, Dijkstra's algorithm is not much better than noise. The average weight feature has very similar results. The results of maximal matching, on the other hand, provide no information on their own in the classical vs. jazz classification problem but are fairly effective in the composer classification.

Conclusion

Overall, the results of both classification problems are exceptional. While binary classification is not a particularly difficult task, scoring higher than 90% accuracy on both tasks is significant, especially with fairly small datasets. The high accuracy suggests that the graph representation of music as well as the set of extracted features are very useful in music classification problems. Future work can apply deep models that utilize these features in classification, using a similar approach to Kenwoo Choi et all [9]. This could understand non-linear relationships between features better and result in higher performance.

Another MIR application where this graph representation may also be effective is music generation. Each graph can be interpreted in some sense as a generative model. At each vertex, the edge weights represent conditional probabilities, where higher edge weights indicate a higher probability that the adjacent vertex, corresponding to a specific musical note, will be played. Once the next vertex is selected, its edges become new conditional probabilities. This interpretation could generate music that mimics the style of certain songs, composers, and even

genres. A difficult challenge will be how to effectively combine multiple graphs into one.

While the graph was quite successful on the classification challenge, there are several ways in which it might be improved. One idea is to generate multiple graphs for the same song. Since songs change over time, capturing the temporal relationship between notes is important. In preliminary tests, using multiple graphs for a song did not significantly improve the performance in classification tasks. However, this may be an important step when extending the graph to music generation as described above.

Other areas for improvement align with how the graph is defined. As mentioned in the introduction, musical rests and note velocity are not considered in the graph. This additional information might make features more effective by capturing more information about how a song is composed and played. On the other hand, including these features would increase the size of each graph. Furthermore, finding Midi data with appropriate velocity information may present a larger challenge to aquiring a sufficient dataset.

Ultimately, this results in this paper are exciting and suggest that this new graph representation of music can be an effective tool in MIR. However, in the broad field of MIR, there is still much to explore.

Figures

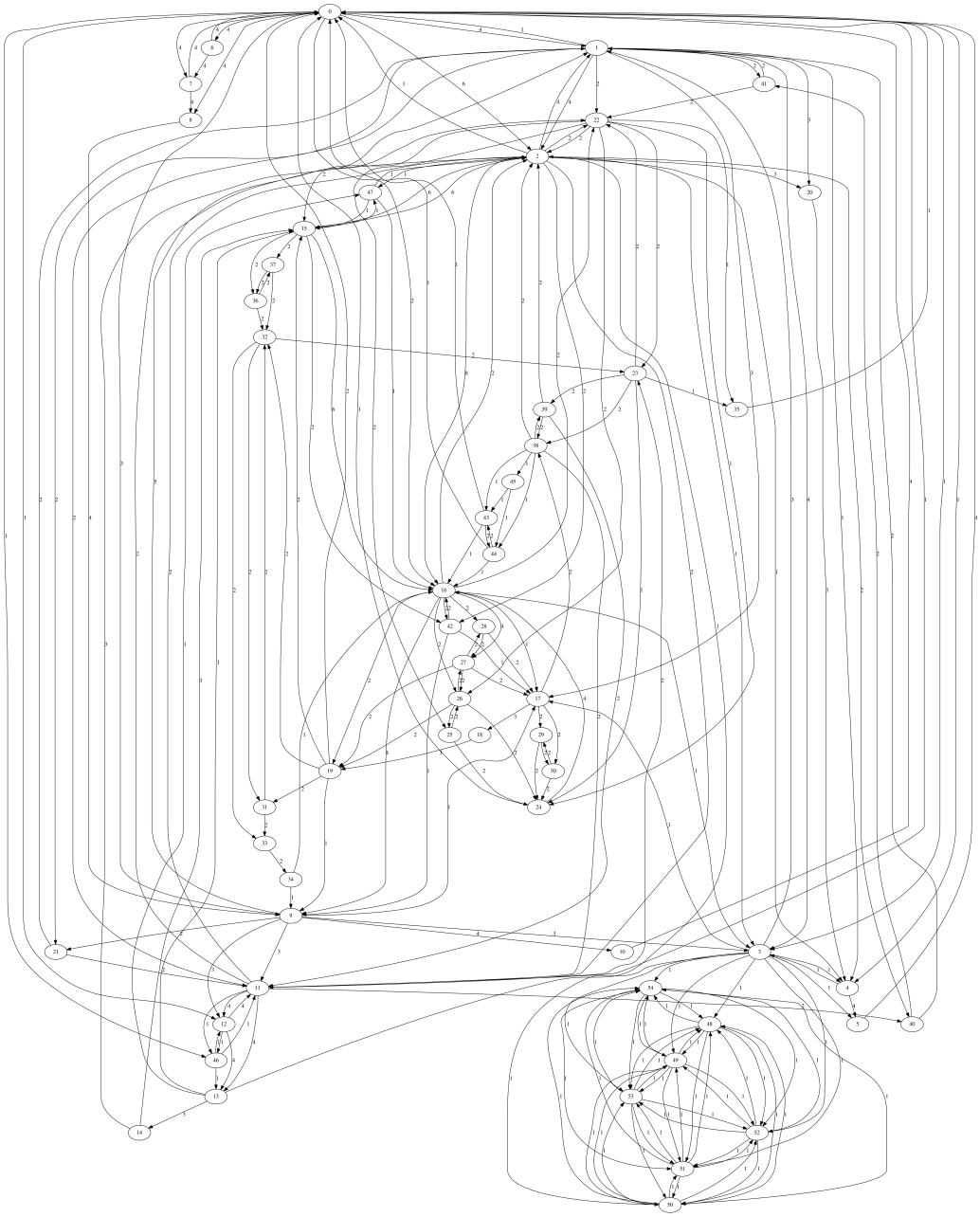

Figure 1: Digraph for Classical MIDI file

+

References

[1] Bernd Krueger. Classical piano midi page, 2016.

[2] Doug McKenzie. Doug mckenzie jazz piano: Video, midi and transcriptions, 2018.

[3] Segundo G. Yogore et al. Kunstderfuge, 2017.

[4] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[5] Alberto Pinto, Reinier H Van Leuken, M Fatih Demirci, Frans Wiering, and Remco C Veltkamp. Indexing music collections through graph spectra. In *Proc. of the 8th International Conference on Music Information Retrieval (ISMIR'07)*, pages 153–156, 2007.

[6] Amit Tiroshi, Tsvi Kuflik, Shlomo Berkovsky, and Mohamed Ali Kâafar. Graph based recommendations: From data representation to feature extraction and application. *CoRR*, abs/1707.01250, 2017.

[7] Z. Fu, G. Lu, K. M. Ting, and D. Zhang. A survey of audio-based music classification and annotation. *IEEE Transactions on Multimedia*, 13(2):303–319, April 2011.

[8] Michael Haggblade, Yang Hong, and Kenny Kao. Music genre classification. 2011.

[9] Keunwoo Choi, György Fazekas, Kyunghyun Cho, and Mark B. Sandler. A tutorial on deep learning for music information retrieval. *CoRR*, abs/1709.04396, 2017.

Python Implementations

Midi to graph parser

```
import mido as M
from mido import MidiFile
from mido import Parser

from sys import argv
import os.path
import numpy as np

#quantize note lengths appropriately
def quantize(note_length):
    lengths = [.03125, .0416, .0625, .0833, .125, .1667, .25, .3333, .5, .75,\
                1, 1.25, 1.5, 1.75, 2, 2.5, 2.5, 2.75, 3, 3.5, 4, 4.5, 5, 6, 7,\
                8, 9, 10, 12, 16,24]

    diffs = np.zeros((len(lengths)))
    for i, l in enumerate(lengths):
        diffs[i] = abs(l - note_length)
    closest = min(diffs)
    return list(diffs).index(closest)

#parse note data from track to list of notes
def parse_note_data(track, ticks_per_beat):
    note_data  = []
    being_played = {}
    tempo = 1
    beats_per_measure = 4
    unit_length = 4
    current_time = 0
    note_counter = 0
    for msg in track:
        current_time += msg.time
        if msg.type   == 'note_on':
            if msg.velocity != 0: #note pressed
                note_counter += 1
                note_data.append([])
                if msg.time != 0: #rest since previous note
                    pass #being_played[msg.note] = msg.time/ticks_per_beat
                being_played[msg.note] = current_time
            elif msg.velocity == 0 and msg.note in being_played: #note released
                duration = current_time - being_played[msg.note]
                note_data[note_counter - 1].append((msg.note, quantize(duration/ticks_per_beat)
                del being_played[msg.note]
        elif msg.type == 'note_off':
            pass
        elif msg.type == 'set_tempo':
```

```
            tempo = msg.tempo
        elif msg.type == 'time_signature':
            beats_per_measure = msg.numerator
            unit_length = msg.denominator
        else:
            pass
    return list(filter(([]).__ne__,note_data))

def generate_graph(note_data):
    if len(note_data) == 0:
        return []

    notes = {}
    note_index = 0
    for chord in note_data:
        for note in chord:
            if note not in notes:
                notes[note] = note_index
                note_index += 1
    adjMatrix = np.zeros((len(notes), len(notes)))

    prev_notes = []
    for chord in note_data:
        for note in chord:
            for prev in prev_notes:
                adjMatrix[notes[prev]][notes[note]] += 1
            for other in chord:
                if note != other:
                    adjMatrix[notes[note]][notes[other]] += 1

        prev_notes = chord
    return adjMatrix

def write_graph(adjMatrix, directory, filename):
    if adjMatrix == []:
        return
    fullname = os.path.join(directory, filename + ".grph")
    fullname = directory + "/" + filename
    outfile = open(fullname, 'w')
    for row in range(len(adjMatrix)):
        for col in range(len(adjMatrix)):
            if adjMatrix[row][col] != 0:
                outfile.write(str(col) + ',' + str(adjMatrix[row][col]) + " ")
        outfile.write("\n")
    outfile.close()
```

```
################################MAIN PROCESS#######################################
output_directory = ""
midi_files = []
for arg in argv[1:]:
    if ".mid" in arg or ".MID" in arg:
        try:
            midi_files.append((MidiFile(arg), arg))
        except:
            print("Unable to open midi file " + arg)
    elif "--dir=" in arg:
        output_directory = arg.replace("--dir=", "")
    else:
        print("Unexpected argument: " + arg)
        print("Expecting midi file (.mid, .MID) or outputfile path")

if output_directory == "":
    print('Not output directory specified')
    quit()

note_data = []
for i, midi_file in enumerate(midi_files):
    mid, name = midi_file
    merged = M.merge_tracks(mid.tracks)
    note_data.append(parse_note_data(merged, mid.ticks_per_beat))
    adjMatrix = generate_graph(note_data[i])
    write_graph(adjMatrix, output_directory, str(i))
    print('Wrote ' + name)
```

Feature extraction process

```
from sys import argv, stdout
import numpy as np
from heapq import heappush, heappop

def matching(adjMat):
    numNodes = len(adjMat)

    dNodes = [sum(row) for row in adjMat]
    nodes = [(i, dNodes[i]) for i in range(len(adjMat))]
    numExposed = len(adjMat)
    matching = [-1 for i in range(len(adjMat))]

    for i in range(len(adjMat)):
            node, unused = nodes[i]
            neighborColors = set()
            if matching[node] == -1:
                for j in range(len(adjMat)):
                    neighbor, unused = nodes[j]
                    if adjMat[node][neighbor] != 0:
                        if matching[neighbor] == -1:
                            matching[node] = neighbor
                            matching[neighbor] = node
                            numExposed -= 2
                            break

    return(numExposed)

def kruskals(adjMat):
    def same_tree(node, neighbor, trees):
        return trees[node] == trees[neighbor]

    def combine_trees(tree1, tree2, trees):
            trees[tree1] = trees[tree1].union(trees[tree2]) #combine trees
            for node in trees[tree1]:
                    trees[node] = trees[tree1]

    numNodes = len(adjMat)

    treeEdges = []
    treeNodes = {}
    for i in range(numNodes):
            treeNodes[i] = {i}

    edgeHeap = []
```

```
    # sort edges
    for node in range(numNodes):
            for neighbor in range(numNodes):
                    if neighbor > node:
                            break; #assume simple, undirected graph
                    edge = adjMat[node][neighbor]
                    if edge > 0: #nonzero weight
                            heappush(edgeHeap, (edge, node, neighbor))

    edge, node, neighbor = heappop(edgeHeap)
    while len(edgeHeap) > 0:
            combine_trees(node, neighbor, treeNodes)
            treeEdges.append((node, neighbor, edge))

            while same_tree(node, neighbor, treeNodes):
                    try:
                            edge, node, neighbor = heappop(edgeHeap)
                    except:
                            break

    treeMatrix = np.zeros((numNodes, numNodes))

    for pair in treeEdges:
            node, neighbor, weight = pair
            treeMatrix[node][neighbor] = weight
            treeMatrix[neighbor][node] = weight

    return(np.sum(treeMatrix))

def dijkstras(adjMat):
    numNodes = len(adjMat)
    root = 0 #defined by problem
    paths = [[] for i in range(numNodes)]
    distances = [-1 for i in range(numNodes)] #-1 signifies infinite distance
    unvisitedNodes = set(i for i in range(numNodes))
    distances[root] = 0

    while (len(unvisitedNodes) > 0):
            unvisitedNodes.remove(root)
            for neighbor in range(numNodes):
                    weight = adjMat[root][neighbor]
                    distance = weight + distances[root]
                    if weight > 0 and (distance < distances[neighbor] or distances[neighbor] ==
                            distances[neighbor] = distance
                            paths[neighbor] = paths[root][:]
                            paths[neighbor].append(root + 1) #nodes indexed by 1
```

```
            try:
                    root = min(node for node in unvisitedNodes if distances[node] != -1)
            except:
                    break #graph is not connected

    return np.mean(distances)

#run Tarjan's Algorithm to count strongly connected components
def num_SCCs(adjMat):
    global generateIndex
    dimen = len(adjMat)

    SCCs = []
    nodeStack = []
    onStack = np.zeros((dimen), dtype='int32')
    generations = np.zeros((dimen), dtype='int32')
    labels = [i for i in range(dimen)]

    inSCC = np.zeros((dimen), dtype='int32')
    generateIndex = 1 #start from 1 as 0 signifies undefined

    def tarjans(currentNode):
        global generateIndex #use same for all function calls

        generations[currentNode] = generateIndex
        labels[currentNode] = generateIndex
        generateIndex += 1

        nodeStack.append(currentNode)
        onStack[currentNode] = True

        #Recurse on neighbors
        for neighbor in range(dimen):
            if adjMat[currentNode][neighbor]:
                if generations[neighbor] == 0:
                    tarjans(neighbor)
                    labels[currentNode] = min(labels[currentNode], generations[neighbor])
                elif onStack[neighbor]:
                    labels[currentNode] = min(labels[currentNode], generations[neighbor])

        if labels[currentNode] == generations[currentNode]: #new SCC found
            SCC = []
            componentNode = nodeStack.pop()
            onStack[componentNode] = False
            SCC.append(componentNode)
```

```
            while componentNode != currentNode:
                componentNode = nodeStack.pop()
                onStack[componentNode] = False
                SCC.append(componentNode)
            SCCs.append(SCC)

    for node in range(dimen): #Run tarjans
        if generations[node] == 0:
            tarjans(node)

    largest = max(len(SCC) for SCC in SCCs)

    return len(SCCs), largest

################################ READ GRAPH DATA ###################################
label = ''
path  = ''
rawdata = []
training = False
for arg in argv[1:]:
    if arg == '--train':
            training = True
    elif "--label=" not in arg:
        rawdata.append(open(arg).read().split('\n'))
    elif "--label=" in arg:
        if label == '':
            label = arg.replace('--label=', '')
        else:
            print("Err: label already specified")
            quit()
    else: #should not happen right now
        print('Unexpected argument: ' + arg)
        print('Expecting .grph file, --label= or --path=')

if label == '':
    print("Err: label not specified")
    quit()

features = np.zeros((len(rawdata), 10))
for i, graph in enumerate(rawdata):
    adjMatrix = np.zeros((len(graph),len(graph)), dtype='int32')
    for r, row in enumerate(graph):
        row = row.split(' ')
        for pair in row:
            split = pair.split(',')
            if len(split) < 2:
```

```
                break #done reading row
            adjMatrix[r][int(split[0])] = int(float(split[1]))

    features[i][0] = len(graph) #Number of nodes
    features[i][1] = np.sum(adjMatrix) #Number of edges
    features[i][2], features[i][3] = (num_SCCs(adjMatrix))
    features[i][4] = dijkstras(adjMatrix)
    features[i][5] = matching(adjMatrix)
    features[i][6] = np.mean(adjMatrix)
    features[i][7] = np.matrix.trace(adjMatrix)

    print(".", end="")
    stdout.flush()
print()

if training:
    outfile = open('data_label_train_' + label + ".csv", 'w')
else:
    outfile = open('data_label_test_' + label + ".csv", 'w')

for row in features:
    for feature in row:
        outfile.write(str(feature))
        outfile.write(',')
    outfile.write(label)
    outfile.write('\n')
outfile.close()
```

Additional code that may be useful for replicating the results can be found on Github (`https://www.github.com/sebasscoates/graphMIR`). All graph data can be found in the same repository.