GSoC 2019: Recording Similarity Index with Annoy for AcousticBrainz

Personal Information

- Name: Aidan Lawford-Wickham
- MusicBrainz, JIRA ticket system, MetaBrainz discourse: aidanlw17
- IRC nick: aidanlw17
- Email: a.lawfordwickham@mail.utoronto.ca
- Github: https://github.com/aidanlw17
- Website & Blog: https://www.noodlab.com
- Twitter: https://twitter.com/AidanLW
- Time Zone: UTC -4

Table of Contents

Overview of the Opportunity

The AcousticBrainz database contains detailed high and low-level information for millions of audio recordings, all of which create an essential for creatives, researchers, and music fanatics alike. Our understanding of audio can be greatly improved through features that focus on similarities between the content of recordings in such a large database. As such, the development of a similarity index between recordings is essential to improving the AcousticBrainz platform and also to the progression of music recommendation engines in related projects like ListenBrainz.

Identifying similarity between recordings is a complex and rewarding project, which has been the subject of many research papers and projects throughout the music industry in recent years. Especially in relation to AcousticBrainz, previous investigations on similarity systems like that of Philip Tovstogan [1], have supported the success of content-based (high and low level data) engines for determining track similarity. These implementations have fallen short since their architecture prevents scalability, ultimately lacking the speed required for use in AcousticBrainz.

With the information gained from previous pitfalls in recording similarity research and the importance of improved efficiency for a long term implementation, my 2019 GSoC project will primarily aim to lay the foundation for an AcousticBrainz similarity engine. The following high level objectives will form a basis for the investigation, ultimately leading to a fast, scalable, and easily updatable system for content-based similarity data on all recordings in the AcousticBrainz database.

High Level Objectives

1. Implement a proof of concept for similarity calculation using the Annoy algorithm.

2. Undergo a series of experiments and benchmarks to determine the validity of the Annoy solution in comparison to other algorithms and the previous implementation [2].

3. Use the proof of concept to build a scalable similarity index for AcousticBrainz.

4. Develop a mechanism to update the index when new recordings are added to AcousticBrainz.

5. Create an API for users and other projects to interact with the similarity index.

6. Use the index to make an offline matrix of similarity for use in other projects.

7. Develop a mechanism to update the offline matrix at a given time interval, when recordings have been added to the database.

Detailed Implementation

Annoy Proof of Concept

Overview:

Using the database alterations and calculations for vectorized metrics from Tovstogan's previous approach [2], we will use the Annoy nearest neighbours software to achieve the same similarity results as Tovstogan's PostgreSQL system. This will verify that Annoy can also accurately provide similarity calculations.

Rationale:

The Annoy nearest neighbours software [3] is expected to be significantly faster than Tovstogan's previous implementation with PostgreSQL. Annoy should improve upon our past issues with this task because it is optimized for memory usage and it uses static files as indexes, distributable between CPU's without ever being rebuilt. This has been largely beneficial for creating the recommendation engine at Spotify, with which our task shares concerns like working with millions of recordings simultaneously. In order to explore this method of determining similarity, we will need to validate that it is able to perform distance calculations between our recordings correctly. We can use Tovstogan's metrics for producing vector ratings from the high and low-level data of each recording, then apply the Annoy algorithm and compare the similarity results to those of his PostgreSQL Cube extension solution.

Details:

In order to implement the Annoy proof of concept, we will need to take the following steps:

1. Gain an understanding of the algorithmic concepts behind Annoy and its Python API.

The first step in implementing a proof of concept is to gain an understanding of the software at a deeper level. Erik Bernhardsson's blog post [4] on the nearest neighbours algorithm driving Annoy provides an appropriate explanation. In summary:

- Annoy's speed depends on dimensionality reduction to create compact representations of otherwise very large vector spaces. This is beneficial for use in the large AcousticBrainz database of recordings.
- Annoy recursively forms a binary tree by randomly selecting two items, and splitting the dataset at an equidistant point between them, then repeating on the two nodes formed until there are at most K elements in each node.

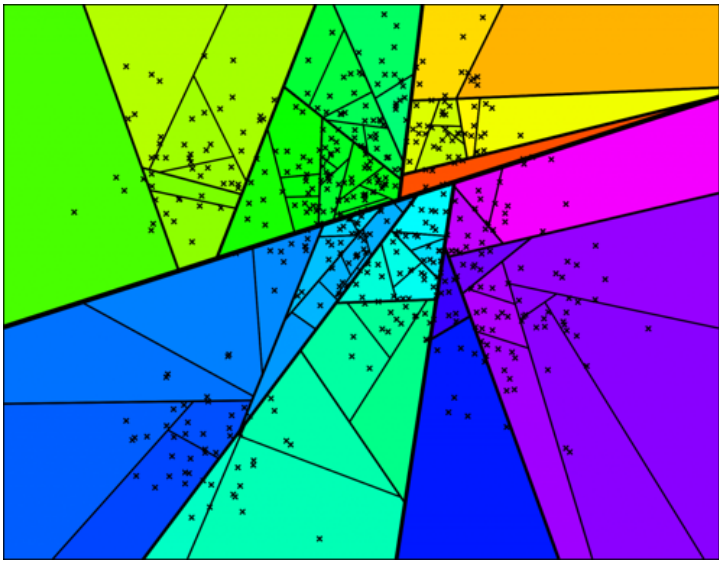


Fig.1: Recursive splits between items that form the binary tree.

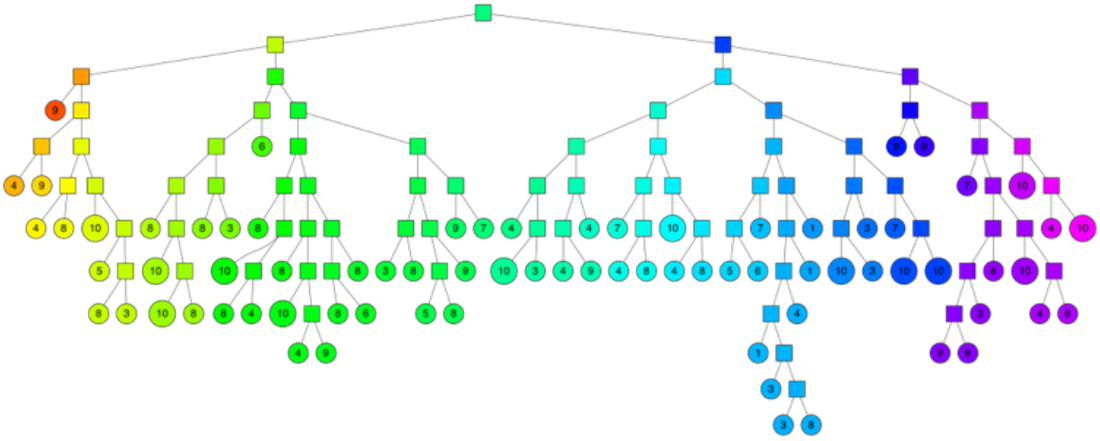


Fig.2: Binary tree formed by the respective splits.

- Notably, each node on the last level of the tree contains a section with at most K items that are near to one another, which can be queried in O(logn) since only depth is traversed.
- With the use of a priority queue to perform these random splits with a forest of multiple trees, accuracy of the nearest neighbours is improved.
- The union of nearest neighbours from each tree is sorted, and the K nearest neighbours to an item (or recording) are returned.

The Annoy python API will be used for our calculations of similarity. Important things to note about working with Annoy:

- `AnnoyIndex(f, metric='angular')` will allow us to create an index of dimension f, with the specified distance calculation method of "metric": "angular", "euclidean", "manhattan", "hamming", or "dot".
- We can add items to the set complete with an integer identifier, and an associated vector. These vectors will be the vectorized ratings created for each recording in the dataset using Tovstogan's work.
- The number of trees used in querying can be predetermined, with more trees improving accuracy but decreasing speed.
- We can query the index for nearest neighbours by identifier or vector, control the number of neighbours returned, and also query for distance between two specific items.

2. Replicating results and understanding the workflow behind Tovstogan's system

Before building the concept, we must also gain an understanding of Tovstogan's methods so that we may replicate his results in comparison to the Annoy system.

This system calculates metrics for all high and low level data related to a recording, such as bpm, key, MFCCs, unary classifiers, moods, instruments, genre models, etc. There are a total of 12 metrics that provide a vectorized rating for each recording. With the calculation of these metrics, this approach utilizes the PostgreSQL Cube extension to index the recordings with a K nearest neighbours approach. The changes that must be made to the acoustic-brainz server in order to reproduce this implementation are viewable here [5].

3. Creating the Annoy proof of concept

We will calculate the metrics used in Tovstogan's approach for each recording in the dataset being used, then add the recording to the Annoy index using its Python API. As mentioned previously, items can be added to the index using an integer identifier and the associated vector. In this case, the calculated metric will be the vector. Notably, the index only allows for identifiers to be integers so the identifier used will be the serial id of the lowlevel table for the recording MBID, which is already tracked and unique to a recording. If this is not preferred, an alternative to track ids for use in the similarity index is to create a reversible hash function which hashes the MBID to a unique, positive integer. Once all the recording items are added, the index can be built and queried for the nearest neighbours of a particular recording. The following pseudocode shows an implementation using each of the metrics in turn, with a similar workflow to Tovstogan's test approach in acousticbrainz-server/similarity/script.py [2]:

```
#Indexing based on one of the 12 metrics
from annoy import AnnoyIndex

mbids = retrieve all mbids for test from database

Create AnnoyIndex object
annoy_index = AnnoyIndex(length of vectors)

with db.engine.connect() as connection:
    #Using Tovstogan's api to retrieve all metrics
    metric_names = metrics.BASE_METRICS #list of metrics
    for name in metric_names:
        metric_cls = metric_names[name]
        metric = metric_cls(connection)
        metric.create()
"""
get mbids - number of ids batch_size with similar db query to
_get_recordings_without_similarity() from Tovstogan's api
"""
       mbids = _get_recordings_without_similarity(connection, name,
batch_size)

        for mbid in mbids:
            #get data for mbid related to metric
            for row_id, data in metric.get_data_batch(mbids):
                #get vectorized data for metric for each mbid
                vector = metric.transform(data)
                index = retrieve id for mbid

                #add item to AnnoyIndex
                annoy_index.add_item(index, vector)

"""all items added to AnnoyIndex, time to build the index with a number of
trees"""
annoy_index.build(number of trees)

# we may save index
annoy_index.save('similarity_index.ann')
# we can load later or transfer and load on another CPU
annoy_index.load('similarity_index.ann')

""" retrieve nearest neighbours from index given an id or index. The below
```

```
is simply a sample and we may work with the nearest neighbours in many ways
for validation. """
idx = id for recording for which we wish to find nearest neighbours
#returns 1000 nearest neighbours
neighbours  = annoy_index.get_nns_by_index(idx, 1000)
```

Testing and Experimentation using Annoy Proof of Concept

Overview:

Once the proof of concept has been developed, it will undergo a series of tests that look to validate the accuracy and efficiency of its similarity calculations between tracks. Experimentation will look to deeply understand how the results from Annoy change with different subsets of the AcousticBrainz database and different types of queries. It will also consist of a comparative investigation between Annoy, the previous PostgreSQL solution, and other nearest neighbours algorithms.

Rationale:

Testing the concept is essential to determining empirically and objectively the ways in which Annoy can most benefit music similarity systems for AcousticBrainz and related projects. Moreover, experimenting with other algorithms on the same dataset and other datasets will allow for us to determine whether or not Annoy is truly the best solution for us to work with. If need be, we can then progress with an alternative algorithm using a similar workflow.

Details:

1. Varied Queries in Annoy and PostgreSQL

We will perform a series of tests that explicitly compare PostgreSQL and Annoy, in order to verify that the use of Annoy is an improvement over the past implementation in terms of speed, and also to verify that Annoy calculates at least as accurately as previous works. During these tests, we will use random sample sets of recordings from the database. The variables in these tests will be the following:

- Size of datasets being indexed
    - We will work with a large range of dataset sizes, allowing us to understand the limitations of each system.
    - Dataset sizes (number of recordings): 10 000, 100 000, 500 000, 1 000 000, full AcousticBrainz database

- Number of neighbours calculated
    - Along with changing the size of the dataset, another factor is the number of nearest neighbours for which we query.
    - Ideally we reach up to 1 000 neighbours, so we will use 100, 500, 1 000, and 2 000 as an appropriate testing range.

- Methods of measuring distance in Annoy
    - The PostgreSQL system utilizes different methods of distance calculation for each metric, however Annoy gives simple flexibility in terms of which distance measurement we use on an index. Varying the distance measurement can change the accuracy of our similarity results, and should also be investigated.
    - The different methods are mentioned under “Annoy Proof of Concept – Details – 1.”

- Number of trees used in building the index
    - Increasing the number of trees improves accuracy of the nearest neighbours, however it will be detrimental to speed.
    - We must find an optimal balance for our query needs. We can test 5, 10, 25, and 50 tree builds.

For a set of tests, each of the above cases will be considered the independent variable whilst time is measured. The time for a query can be measured simply using the time library as follows:

```
import time

start_time = time.time()

#perform query

end_time = time.time()

query_time = end_time - start_time
```

We must also test accuracy and so the nearest neighbours returned for each test case above can be compared between the Annoy and PostgreSQL methods. This check can return the number of discrepancies in the nearest neighbours array for each query.

It will be valuable to organize the experimental data visually to gain a holistic understanding of the results. We can organize the results into a table in the following manner:

Metric	K (# nearest neighbours)	Size of Dataset	# Discrepancies in Neighbours	Annoy Distance Method	# Trees in Annoy	Time Annoy	Time PSQL
...	...					...	...
…	...					...	...

Plots can also be produced easily via matplotlib or other software. The following plots of the results will be beneficial:

With data for Annoy and PostgreSQL on the same pair of axes for each metric:
- K vs Time
- Dataset size vs Time
- # Trees in Annoy vs # Discrepancies
- Annoy Distance Method vs # Discrepancies

Solely for Annoy for each metric:
- # Trees vs Time
- Dataset size vs Time
- Distance method vs Time

2. Utilizing ANN Benchmark Comparisons

It would also be highly beneficial to understand the results of Annoy in comparison to other nearest neighbours algorithms as seen in Erik Bernhardssons benchmark testing [6], which provides an empirical evaluation of with a variety of algorithms on numerous datasets. We can create a fork of this repository and using the metrics calculations generated in Tovstogan's past work, generate alternative datasets using the metrics associated with each recording. Once we format our data to HDF5, we can test Annoy against other nearest neighbours algorithms on our dataset to see their differences in performance. The results of this benchmark testing will be the recall queries per second tradeoff, in which a higher number per second is preferred.

3. Formatting a Report and Documenting Results

After performing this series of tests, it will be crucial to properly document the methodology used and the results of the testing. This way, it can be replicated for improvements and other projects in the future, and we can verify the conclusions that we draw.

Similarity Index System with Continuous Update and API

Overview:

After verifying the use of Annoy for recording similarity via a process of experimentation (or selecting an alternative nearest neighbours algorithm), the next step is to create a long term strategy to implement the index in AcousticBrainz. This system will need to be updated continuously as new recordings are added to the database, and it will also require a Flask API so that users can query the similarity of recordings using the new index.

Rationale:

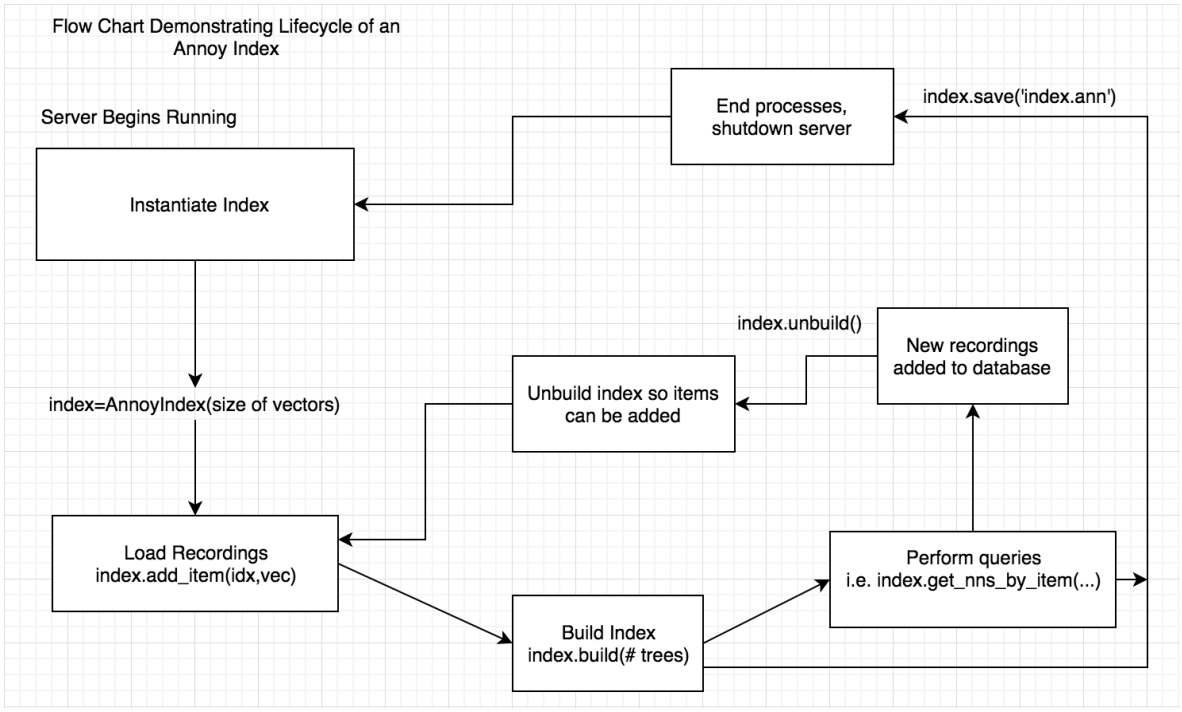
An Annoy index implementation in AcousticBrainz allows for other features to be built on recording similarity, and allows for AcousticBrainz users to actually access the similarity engine capabilities.

Details:

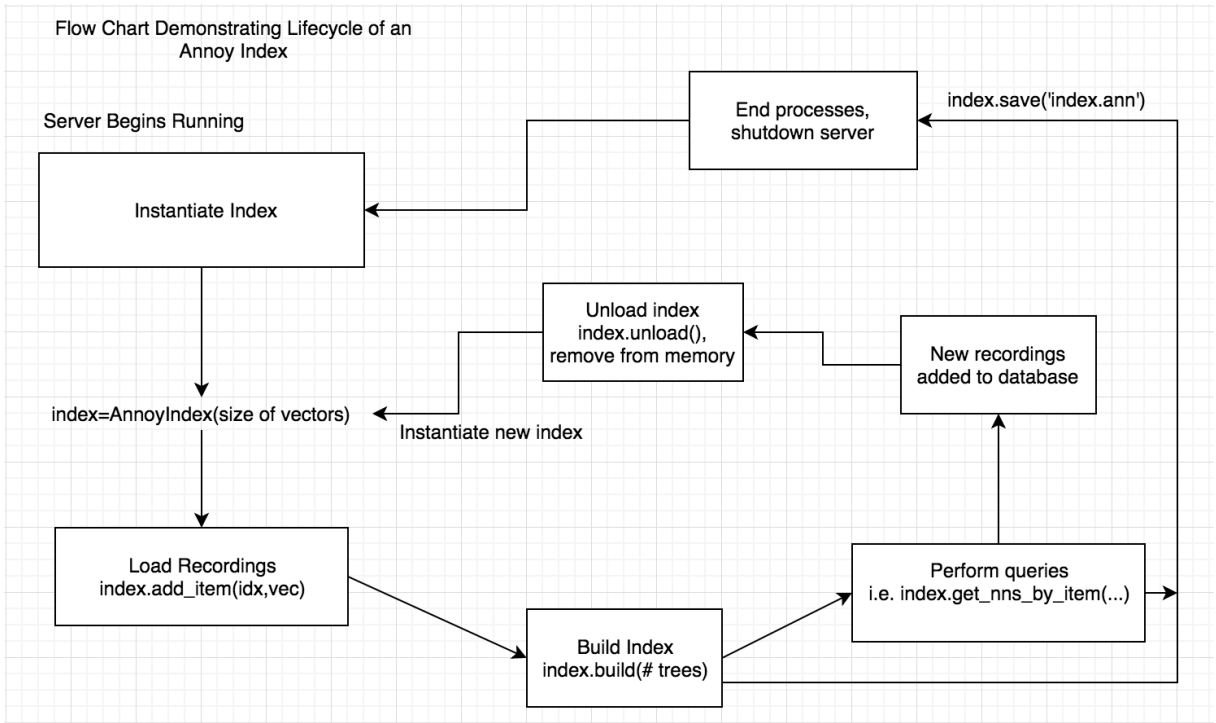
1. Similarity Index and Update Mechanism Workflow

At a high level, the system for creating a similarity index will involve creating multiple Annoy Indices, specifically one for each metric in use. When the server begins running, new Annoy Index objects will be created and all recordings will be added as items before the indices are built and saved. After the first time that this occurs, the indices will be loaded from their saved state rather than being rebuilt time and time again. A listener will be implemented on the acousticbrainz-server functions that allow for new recordings to be added to the database. When these functions are called and new recordings are added, the respective indices will be unloaded and removed, and newly created indices that add all items from the database will include the new recordings.The order of these operations is not certain, i.e. it may be advantageous for us to create new indices, removing the existing indices only once this process is finished - allowing indices to be accessible by users at all times.

Note: the process of updating the indices is complicated since the current state of Annoy does not allow for new items to be added to an existing index once it is loaded, saved, or built. This means that we need to recreate indices with each added recording. Given the size of our database this may be slow, and it may be possible to speed this up with some modifications to Annoy [7]. This involves adding an `unbuild()` function to Annoy which would allow items to be added to the index after it has been built. This is still limiting in that we are required to rebuild each time the server begins running, but with added unbuild functionality we can greatly improve the update mechanism as follows:



Conversely, without `unbuild()` functionality:



Ideally, we increase the efficiency of this implementation by adding the unbuild functionality as proposed in [7].

2. Similarity Index Class Model

The following file trees show the required files for adding this index functionality:

```
acousticbrainz-server
    webserver
        views
            api
                v1
                    __init__.py
                    core.py
                    dataset_eval.py
                    datasets.py
                    similarity.py
```

And

```
acousticbrainz-server
    similarity
         manage.py
         annoy_model.py
         metrics.py
         operations.py
```

The `acousticbrainz-server/similarity` folder holds the implementation of the Annoy Index using a wrapper class for the index in `annoy_model.py`. Note that the `metrics.py` and `operations.py` have been copied from Tovstogan's implementation [2] for the calculation of the required metrics, and the associated alterations to the database via files in `acousticbrainz-server/admin` will also need to be made.

The file `annoy_model.py` contains the class AnnoyModel, which wraps the Annoy Index and has methods for all functionality required to interact with the Annoy Index. This makes it very simple for users to interact with the Annoy Index and for features to be built on the index. The class will contain the following methods:

- `__init__(self, num_trees, metric, distance_type)`
    - Initialize model with a given number of trees, a keyword for which metric it indexes with, and `distance_type` indicating the method of calculating distance.

- `add_items(self, items)`
  - Adds items to the index if it is initialized. Items is a list of tuples to be added, in which the first value in a tuple is the identifier for a recording and the second value is the vector related to the metric that we look to calculate.
  - Metric calculations to get vectors are applied based on Tovstogan's work as in the aforementioned proof of concept.

- `build(self)`
  - Builds the index with the number of trees specified for the index. Error checks the build process, returning True for a successful build and False with error messages.

- `unbuild(self)`
  - Uses unbuild functionality that is added to Annoy, unbuilds the index so that new items can be added. Returns `True` for a successful unbuild and False for errors with error messages.

- `save(self, name, base_path)`
  - Saves the index to the disc under the name `'name.ann'` with the path `base_path + 'name.ann'`.

- `load(self, path)`
  - Loads the index with the specified path.

- `get_nns_by_id(self, id, num_neighbours, distances)`
  - Queries the index for the nearest neighbours to the recording with the specified id.
  - Number of neighbours returned is `num_neighbours`, and the function returns a list of the neighbours or a list of tuples in which the first element of a tuple is the neighbour and the second is the distance, if `distances=True`.

- `get_nns_by_vector(self, vector, num_neighbours, distances)`
  - Same query as `get_nns_by_id`, except via the use of a metric vector to search rather than an identifier for the recording.

- `get_item_vector(self, id)`
  - Returns the vector for the index's metric that is associated with the given recording identifier `'id'`

- `get_distance(self, idA, idB)`
  - Returns the distance between the vectors associated with identifier `'idA'` and `'idB'`.

- `get_id(self,mbid)`
    - Given an MBID, this method queries the database for the related integer identifier that is used in the Annoy queries, returning this integer.

3. Flask API for Users to Interact with the Index

`acousticbrainz-server/webserver/views/api/v1/similarity.py` will add an API for similarity queries on the AcousticBrainz database. The API methods will make use of the existing similarity indexes that are created by Annoy, and the `annoy_model.py` class to make queries. It will include the following GET requests:

```
bp_similarity = Blueprint('api_v1_similarity', __name__)
```

```
@bp_similarity.route("/nns/metric/<uuid:mbid>", methods=["GET"])
```

Parameters: MBID, num_neighbours, metric, full_json
Response Headers - Content-Type: application/json

- `get_nns_by_mbid(mbid, num_neighbours, metric, full_json)`
    - `num_neighbours` will determine the number of neighbours that are returned in by the query.
    - The index used for the query will depend on the metric selected, i.e. there are 12 metrics so the API may query for similarity with respect to any of these different metrics.
    - Nearest neighbours will be calculated for the given mbid. This query is done by MBID rather than the integer identifier related to Annoy for ease of use. The associated integer id will be taken from the database and used in the inner Annoy Index method, thus abstracted from the user.
    - If `full_json=True`, then the query will collect the high and low level data for all nearest neighbours, and return both the mbids of nearest neighbours as well as their JSON data. Else, the query returns only JSON of nearest neighbour ids and associated vectors.

    Sample data response:

```
 { "neighbours":
 [{"id1": id1, "vec1": vec1 }, ... , {"idN": idN, "vecN": vecN}]
}
```

```
@bp_similarity.route("/nns/metric/<uuid: mbid>", methods=["GET"])
```

Parameters: dataset_id, num_neighbours, metric, full_json
Response Headers - Content-Type: application/json

- `get_nns_by_mbid(mbid, num_neighbours, metric, full_json)`
  - Retrieves nearest neighbours by MBID for all mbids present in the dataset, similar operations to that of retrieval for single recording

  Sample data response:

```
{ ["mbid1": {"neighbours":
 [{"id1": id1, "vec1": vec1 }, ... , {"idN": idN, "vecN": vecN}] },

"mbid2": {"neighbours":
 [{"id1": id1, "vec1": vec1 }, ... , {"idN": idN, "vecN": vecN}] },

...

"mbidN": {"neighbours":
 [{"id1": id1, "vec1": vec1 }, ... , {"idN": idN, "vecN": vecN}] }]
}
```

```
@bp_similarity.route("/nns/metric/<uuid: mbidA>/<uuid: mbidB>",
methods=["GET"])
```

Parameters: mbidA, mbidB, metric
Response Headers - Content-Type: application/json

- `get_distance(mbidA, mbidB, metric)`
  - This query returns the distance between the recordings with the given two MBIDs in JSON format. Calculates distance using the index associated with given metric.
  - Sample data response:
    `{ "metric": metric, "distance": distance }`

```
@bp_similarity.route("/nns/metric/<uuid: mbid>/vector", methods=["GET"])
```

Parameters: mbid, metric
Response Headers - Content-Type: application/json

- `get_item_vector(mbid, metric)`
  - Translates the MBID to associated integer id for Annoy, and returns its vector for the given metric in JSON format.
  - Sample data response:
    `{ "metric": metric, "mbid": mbid, "vector": vector }`

Additionally, after creating the above endpoints it would be ideal to implement a user interface to interact with the API. Decidedly, this is not prioritized as highly as creating an offline similarity matrix and may take place after the following activities have been completed, i.e. if there is time in the last two weeks of GSoC or as future work to add to the project after GSoC.

4. Unit Tests and Create Documentation

After implementing the Annoy Index and its API, the added functionality must be tested thoroughly via the introduction of unit tests and clear documentation of the process and its utility.

Offline Similarity Matrix of Recordings with Update System

Overview:

In order to make the similarity index on AcousticBrainz useful for other projects and applications in the future, it would be largely beneficial to be able to store the similarity data offline, in the form of a matrix. Since the data from the similarity matrix is likely to be read and loaded frequently in applications, and speed is prioritized, I believe it would be beneficial to either save the matrix as a native numpy object, using the .npy format, or save it in HDF5 using the h5py library.

In using npy we will benefit from faster read speeds after the first time that the data is loaded. Especially when compared to using other structures like csv for this data, we will save a significant amount of time. The load speed for this file type has been shown to be constant over increasing file sizes, which will be crucial for our application since the database contains millions of recordings. See an experiment on this noted here [8]. Additionally, this allows us to use numpy arrays with ease which are more efficient than regular python objects, and are based on a vectorized implementation.

On the other hand, HDF5 works efficiently with large datasets and allows for us easily to create, share, and analyze our dataset using the same format. With the use of the h5py library, we can easily create numpy arrays for matrices and save them in a file to be used in other applications. We can also access matrices related to individual metrics as datasets from the same file using HDF5, with indexes akin to python dictionary keys.

It will also be necessary to implement a system that updates the offline matrix continuously at a given time interval, since recordings will be constantly added and changes in the database. This way, the offline matrix that other applications depend on can remain current.

Details:

1. Develop Storage Method and Format

It will be beneficial for other applications to gain access to a large sample of similar recordings for each recording in the database. The number of similar recordings held in the matrix may be altered in the future or created on a case-by-case basis, however we will begin by creating a 1001xN matrix for each index, in which N is the number of recordings in the AcousticBrainz database. The similarity matrix will be available for each of the 12 metrics. The shape of the matrix will be the following, with each recording at the top of a column and the 1000 most similar recordings ordered below (note that the leftmost column is for reference, not actually a part of the matrix):

Similar Recordings	MBID1	MBID2	MBID3	…	…	MBIDN
1	Most similar MBID	…	…	…	…	Most similar
…	…	…	…	…	…	
1000	Least similar MBID	…	…	…	…	Least similar

In `acousticbrainz/similarity` we will add the file `matrix_generator.py`. The following pseudocode will describe the process to generate a matrix in the form of a Numpy array for a given index. After generating a numpy array, it can be saved in HDF5 format or using npy:

In `matrix_generator.py`:

```
import numpy as np
import h5py
from annoy_model import AnnoyModel
from metrics import BASE_METRICS

# Define the following within a function, i.e. gen_matrices()
# Generates matrix for each of the metrics

# initialize HDF5 file with h5py
h5f_matrices = h5py.File('similarity_matrices.h5', 'w')
with db.engine.connect() as connection:
    #Using Tovstogan's api to retrieve all metrics
    metric_names = metrics.BASE_METRICS #list of metrics
    for name in metric_names:
        metric_cls = metric_names[name]
        metric = metric_cls(connection)
        metric.create()

        #create AnnoyModel for each metric
        annoy_index = AnnoyModel(# trees, name, distance_type)

"""get mbids - number of ids batch_size with similar db query to
_get_recordings_without_similarity() from Tovstogan's api """

        mbids = _get_recordings_without_similarity(connection, name,
batch_size)
        for mbid in mbids:
            #get data for mbid related to metric
            for row_id, data in metric.get_data_batch(mbids):

                #get vectorized data for metric for each mbid
                vector = metric.transform(data)
                index = retrieve id for MBID
                #add item to AnnoyIndex
                annoy_index.add_item(index, vector)
```

```
# all items added to AnnoyIndex, time to build the index with a number of
trees
annoy_index.build(# trees)
np_similarity_matrix = None
for mbid in mbids:
    index = retrieve id for MBID
    neighbours = annoy_index.get_nns_by_id(index, 1000)

    # change ids to MBIDs
    for i in range(1000):
        neighbours[i] = [query db for mbid with given id]

        neighbours = [[mbid]] + neighbours
        if np_similarity_matrix is None:
            np_similarity_matrix = (np.array(neighbours)).T
            continue
        neighbours_np_array = np.array(neighbours)

        # concatenate columns containing MBIDs
        np_similarity_matrix = np.concat((np_similarity_matrix,
neighbours_np_array.T), axis=1)

matrix_name = name + '_metric_matrix'


# save np_similarity_matrix in .npy format
matrix_name += '.npy'
np.save(matrix_name, np_similarity_matrix)

or

# save np_similarity_matrix in HDF5
h5f_matrices.create_dataset(matrix_name, data = np_similarity_matrix)

# can be loaded with np.load(matrix_name)
or
# h5f_matrices = h5py.File('similarity_matrices.h5', 'r')
# matrix = h5f_matrices[matrix_name]
```

1.1. Testing Storage Methods

Note that we should attempt saving the matrix using both of these file formats, and develop tests using the time library around the saving and loading times when using each of the formats before determining the ideal solution.

2. Develop Update Mechanism

The matrix should be regenerated via the above functionality at a given time interval, only executing if new recordings have been imported. This can be checked via checking a True/False variable within the gen_matrices function:

- Variable `any_new_recordings` set to `True` when recordings imported
- Call gen_matrices when recordings are imported
- Import threading and time libraries
- In beginning of `gen_matrices`, if `any_new_recordings = False`, exit from function.
- Implement `threading.Timer(time interval, gen_matrices).start()` within `gen_matrices`

3. Unit Testing and Documentation

Finally, it will be necessary to write unit tests to ensure the performance of the added functionality related to the offline matrix. This functionality and its utility will also be documented appropriately.

Timeline

High Level Timeline

May 6 – 27: Community Bonding Period

I will spend the community bonding period frequently hanging out on the IRC chat, getting to know my mentors and the surrounding community on a deeper level, and working on other contributions to AcousticBrainz. I will also spend time working with my mentors to sharpen my proposal and focus on further developing and clarifying some key aspects of the project, ensuring that my project's features truly meet the needs of the community. I will also spend time further gaining an understanding of previous theses and research related to recording similarity, and look to begin my implementation of the Annoy Proof of Concept as soon as possible. It will also be beneficial to get Tovstogan's previous implementation running during this period.

May 27 – June 3: **Milestone 1** – Annoy Proof of Concept

In the first week of GSoC, I plan to complete the implementation of a proof of concept for similarity calculations using the Annoy algorithm. Given that I will have already devoted a sufficient period of time to understanding the Annoy algorithm and the existing PostgreSQL implementation metric calculations, I expect to be able to focus on building/finishing the proof of concept in this period. This will involve extending manage.py with functions similar to those written by Tovstogan in similarity/manage.py, implementing changes to the database from the previous work, and utilising the metric calculations to create vectors for use in the Annoy Index. I will also be creating a copy of Tovstogan's work that allows me to test his system against the Annoy concept.

June 3 – June 11: **Milestone 2** – Testing and Experimentation with Annoy Proof of Concept

Once the proof of concept has been implemented, I will dive into further developing the testing methodology and working through the tests that I outlined under this section. I will be developing tests explicitly between Annoy and the PostgreSQL solution, as well as working with the ANN benchmark comparisons to format our dataset and test Annoy against other nearest neighbours algorithms. A significant portion of this section will be dedicated to properly, carefully documenting and reporting on the results of our experimentation. I will look to visualize the results and draw conclusions that will aid future implementations.

June 11 – June 28: **Milestone 3** – Similarity Index System with Continuous Update

Once we have verified the use of Annoy or possibly pivoted to another indexing system with a proof of concept and conclusive testing, I will focus my time on implementing the index in the acousticbrainz-server. I will focus on developing and storing multiple indices for each metric that is used to find similarity between recordings. Furthermore, I will develop a mechanism to continuously update the indices when new recordings are added. This process will require some exploration and possible alteration of the generic Annoy library to allow for us to "unbuild" the index and add new items, improving the efficiency of our update system. It will be crucial for me to develop unit tests related to this implementation and document the processes before moving on to the next task.

July 1 – July 5: **Milestone 4** – Similarity Index API

This section of the project has a lighter load, which is designed to provide some extra time to sort through possible bug fixes or issues that may have caused setbacks in earlier milestones. Aside from working through bug fixes, it will be necessary to implement an API for the new similarity index system, allowing for users to access similarity information. I will work to build the related functions required for the API and develop unit tests for their functionality, as well as clearly document the API for ease of use.

July 8 – July 26: **Milestone 5** – Offline Similarity Matrix of Recordings

During 5th milestone of my GSoC project, I will focus on creating functionality that creates matrices of the most similar recordings to each recording, based on the individual metrics. I will create the function to generate matrices and develop both of the possible storage methods, as well as considering alternatives. With these different methods of storage, I will perform a series of tests related to loading and saving the matrices, ultimately determining the optimal method of storage. I will then shift to implement this method of storage offline and create unit tests for the functionality related to generating and saving matrices.

July 29 – August 16: **Milestone 6** – Update Mechanism for Offline Recordings

As I complete the final milestone of the project, I will focus on implementing a mechanism to generate new matrices based on a timer, as well as indicators that new recordings have been submitted to the database. I will also write unit tests for this functionality, fix bugs, and use this period of time to catch up on any milestones that may have slowed my progress.

August 19 – August 26: Documenting and Deploying

The final week of my GSoC project will be fully dedicated to fixing issues, cleaning up code and unit tests, and ensuring that I have clear documentation for the functionality that I've added. I will spend this time making sure that my implementation is ready to be deployed to the acousticbrainz-server, or taken to the future stages of the project.

Future Work

In the future, I truly look forward to continuing to contribute to AcousticBrainz and especially this project, of which I have already grown quite fond. I will continue to implement related functionality like developing the offline similarity matrix as a data dump. I would also like to consider developing a new hybrid metric that accounts for all other metrics, in a sense looking to add some sense of finality to the similarity data that we receive from all individual metrics. I would then implement the above features with this metric as well. Moreover, I would like to add navigation by similarity to AcousticBrainz during the fall after GSoC. I look forward to becoming a long term contributor to AcousticBrainz and MetaBrainz as a whole!

Week-by-Week Timeline

Week	Task
May 6 - 13	Bonding with mentors, fixing tickets, reviewing previous work and docs.
May 13 - 20	Bonding, implementing Tovstogan's work locally, discussing project.
May 20 - 27	Bonding, clarifying issues with mentors, setup/begin Annoy concept
May 27 - June 3	Finishing Tovstogan's work, creating Annoy proof of concept
June 3 - 10	Test Annoy vs. other solutions, utilise ANN benchmarks, documentation
June 10 - 17	Implement chosen similarity index in acousticbrainz-server
June 17 - 24	Investigate using "unbuild" for update mechanism, build mechanism
June 24 - July 1	Finish index update mechanism, unit tests and documentation
July 1 - 8	Create similarity index API, develop unit tests and document usage
July 8 - 15	Functionality for matrix generation, develop methods of storage
July 15 - 22	Test methods of storage for matrix, time for catch up if required
July 22 - 29	Implement one storage method, unit tests for matrix generation
July 29 - August 5	Add update mechanism functionality for offline matrix
August 5 - 12	Finish update mechanism, unit tests and documentation for mechanism
August 12 - 19	Fix bugs and catch up if necessary, reflect, make possible improvements
August 19 - 26	Checking tests, documentation, and preparing for final submission

Links

[1] https://zenodo.org/record/1479769#.XKy3mZNKg1K

[2] https://github.com/philtgun/acousticbrainz-server/tree/master/similarity

[3] https://github.com/spotify/annoy

[4] https://erikbern.com/2015/10/01/nearest-neighbors-and-vector-models-part-2-how-to-search-in-high-dimensional-spaces.html

[5] https://github.com/metabrainz/acousticbrainz-server/compare/master...philtgun:master

[6] https://github.com/erikbern/ann-benchmarks

[7] https://github.com/spotify/annoy/issues/174

[8] https://github.com/tirthajyoti/Machine-Learning-with-Python/blob/master/Pandas%20and%20Numpy/Numpy_Reading.ipynb

Detailed Information about Myself

My name is Aidan Lawford-Wickham and I am an undergraduate student in Engineering Science at the University of Toronto in Canada. I am currently a prospect for specialization in the Machine Intelligence stream. After discovering the Google Summer of Code, I found myself most interested in the MetaBrainz Foundation because of my passion for music.

Tell us about the computer(s) you have available for working on your SoC project!

-I have a PC with Intel 8th gen i5 8600K processor, gtx 1080 graphics, 16gb RAM. I am dual booting Ubuntu 18.04 and Windows 10.
-When I'm not at my desk, I use the early 2015 Macbook Pro 13 inch model, Intel i5 2.7GHz, Intel Iris Graphics 6100, 8GB RAM. Running macOS Sierra.

When did you first start programming?

I began learning basic frontend web development during high school in grade 10, and quickly introduced myself to Python and Java. In my early days, mobile applications peaked my interest and I attended iD Tech Coding and Engineering Academy at MIT for the summer between grades 10 and 11 to focus on Android app development.

What type of music do you listen to? (Please list a series of MBIDs as examples.)

I love listening to many genres and find inspiration for making my own music in doing so.

Some of my favourite alternative-pop RnB is from Frank Ocean's nostalgia,ULTRA. mixtape:

1ee6d0f9-dedb-43db-a337-44acf5be3832,

d212e3ea-27b7-40ba-a322-524f698d8a37,
37e280a5-f222-4fe8-b544-b4951eabdd5d

I'm deeply interested in rap music and culture, one of my favourite artists is Mac Miller:

e3cb3c99-6125-4ef5-8a23-53268482ca3e,
29cff7c4-b8fb-44cf-91f1-81bcd31c2e3c,
6b757544-f6f8-40b0-8d2c-c5ef9b962ca6

I love listening to melodic guitar riffs, hence blues rock from The Allman Brothers Band:

0b018d6e-0e93-416f-838c-d46da444f7aa,
E9d1d205-4acf-4cd4-b1ea-fb095d065c32

Alternative rock when I'm in the mood - Red Hot Chili Peppers:

4ffc971d-2638-4409-be49-32d1acfe6afd

What aspects of the project you're applying for (e.g., MusicBrainz, AcousticBrainz, etc.) interest you the most?

Given that I am passionate about both music and computer science, finding an area where these two subjects overlap would be a dream for me. I'm most interested in the content-based data that the AcousticBrainz database holds for such a large number of recordings. I'm fascinated by music recommendation systems especially since discovering Spotify's personalized "daily mix" feature, and I'm looking forward to seeing what results we can get from building a similarity index into AcousticBrainz.

Additionally, I'm interested by the community aspect of AcousticBrainz and MetaBrainz as a whole. I'm intrigued by the welcoming, positive atmosphere of this project and I look forward to putting forth my own contributions.

Have you contributed to other Open Source projects? If so, which projects and can we see some of your code?

I have not contributed to other Open Source projects. I have, however, made a pull request to AcousticBrainz to improve the CONTRIBUTING.md documentation, and another to fix ticket AB-387 (Import/Export in CSV with descriptions).

CONTRIBUTING.md - https://github.com/metabrainz/acousticbrainz-server/pull/334

AB-387 - https://github.com/metabrainz/acousticbrainz-server/pull/333

If you have not contributed to open source projects, do you have other code we can look at?

My GitHub profile has repositories for a number of projects that I've worked on.

I have worked with the University of Toronto data science team this year doing research and creating machine learning models to predict energy prices on peak days for Ontario, Canada. This is the repo: https://github.com/aidanlw17/energy-demand-forecasting

I recently built a chess game that plays in your favourite shell, featuring both two player and a mode playing against an AI. The AI is built using alpha-beta pruning and the minimax algorithm: https://github.com/aidanlw17/csc190-chessai/tree/master

This is an instant-chat web application built with Django Channels and a React frontend: https://github.com/aidanlw17/locator-anon

I've built a program to solve an optimization problem for Formula 1 racing. The program uses reinforcement learning techniques to determine the actions (accelerating or braking) of the driver at each point on the race track, and also optimizes the car configuration depending on the track chosen:
https://github.com/aidanlw17/daisy-formula-one

My personal website (admittedly a work in progress, but getting there - do not view on mobile just yet), https://www.noodlab.com, is built using React:
https://github.com/aidanlw17/alawfordwickham_site/tree/master/src

I've also done stock price prediction using a recurrent neural network:
https://github.com/aidanlw17/trading-rnn

An android galactic survival game made in Java:
https://github.com/aidanlw17/AndroidAsteroidsGame

What sorts of programming projects have you done on your own time?

See the above question for some examples. I am very interested in machine learning and data science, particularly in time series data. I have pursued projects related to this area in the past such as working with the University of Toronto data science team and doing a stock prediction project. In terms of data science skills, I have experience working with scikit-learn, numpy, tensorflow and keras, and SQL databases. I'm also interested in web development. Given my data science background, I'm very fond of python and so I've spent time developing backend with Django and Flask. I also enjoy frontend development and have worked with React. I love playing video games and have naturally been drawn to building games like a galactic thriller for

the Android platform. I have experience writing sorting algorithms in C, and worked this year as the embedded systems lead for the Phantom SUMO robot team at the University of Toronto.

How much time do you have available, and how would you plan to use it?

I plan to be working on my GSoC project full time - I expect to put 40+ hours of work into this project weekly, or at least 8 hours a day. I will also be available to work on the weekends whenever necessary. In my spare time during the summer, I will enjoy water sports like sailing and wakesurfing, and playing soccer.

Do you plan to have a job or study during the summer in conjunction with Summer of Code?

Aside from GSoC, I will not be working this summer. I will dedicate some of my spare time on weekends to my side projects to continue developing my skills in machine learning and web development, however I also expect to further develop my skills via my GSoC work.