

# How I used PostgreSQL<sup>®</sup> to find pictures of me at a party

By Tibs (they / he)

Slides and source code at  
<https://github.com/aiven-labs/pgvector-find-faces-talk>

*tony.ibbs@aiven.io / https://aiven.io/tibs / @much\_of\_a*

# Broad structure

Introduction and vague background

Not an explanation of ML

Finding pictures of me

Why PostgreSQL®.

# Part the first: Introduction and vague background

I'm an AI skeptic

I lived through the last **AI boom** in the 1980s, and the subsequent **AI winter** of the 1990s

But we did get expert systems, knowledge based systems, etc. - they just dropped the name "AI"

# My colleagues have been convincing me

- Quick prototypes of boring code
- Rewrite this paragraph a different way
- And now, finding my face

Also, recently, a demo I saw on multi-modal comparisons - comparing text, audio, image and video.

# Part the second: not an explanation of ML



*Photo by [Markus Winkler](#) on [Unsplash](#)*

*tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a*

# Not an introduction to vectors and embeddings

ML people talk about vectors and embeddings and vector embeddings.

"Embedding" means representing something in a computer.

So a "vector embedding" is

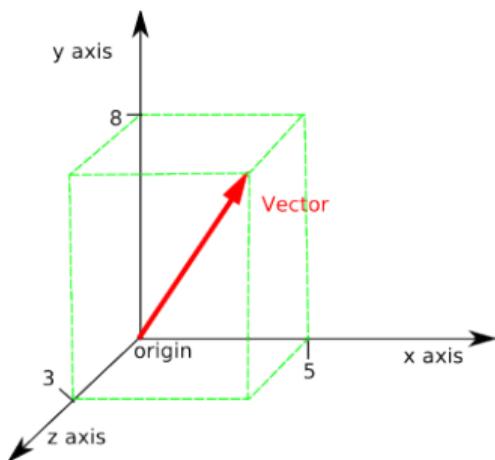
- a vector that represents something,
- stored in a computer.

# Not enough about vectors

Broadly, we can describe the characteristics of things with numbers.

For instance, we can describe colours with RGB values.

# A 3d graph showing a vector



*Image from JCC Math.Phys 191: The vector class, CC BY-SA 3.0*

# We can do mathematics with vectors

We can compare their

- length
- direction

and we can do maths between vectors - but look elsewhere for that

# Calculating the vectors

By hand for relatively simple cases

(for instance, in early text analysis)

but with ML, we can

- *train* a machine learning system
- to "recognise" that a thing belongs to particular categories.

This is wonderful - and sometimes leads to surprising results

# Part the third: Finding pictures of me



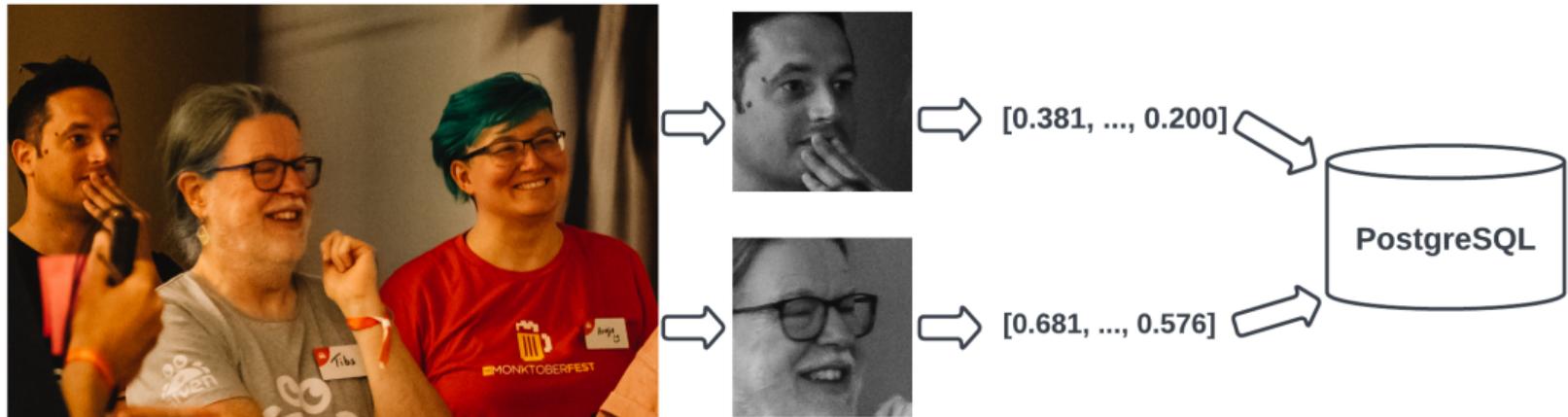
*tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a*

# Our aim

Find which files contain my face, using SQL like the following:

```
SELECT filename FROM pictures
ORDER BY embedding <-> [0.38162553310394287, ..., 0.20030969381332397]
LIMIT 10;
```

# Stage 1. Find faces and store their embeddings



*tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a*

# It's not perfect!

May not find all the faces

When analysing a group photo, it also found these two faces:



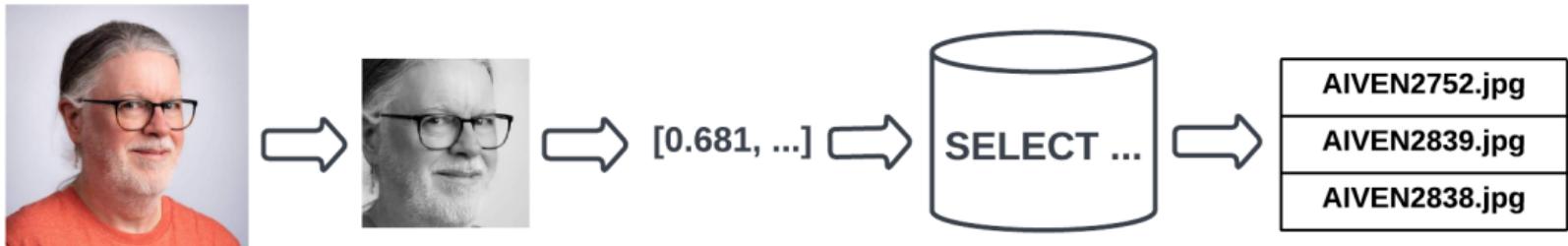
*tony.ibbs@aiven.io / https://aiven.io/tibs / @much\_of\_a*

# 768 floating point numbers

Each embedding is an array of 768 floating point numbers.

0.38162553310394287, ..., 0.20030969381332397

## Stage 2. Look for photos with my face in them



Using my slack image as the reference face

# Set up the environment

We're going to be using

- `opencv-python` to find faces
- `imgbeddings` to calculate embeddings from an image
- the `haarcascade_frontalface_default.xml` file from the [OpenCV GitHub repository](#), which defines the pre-trained Haar Cascade model

My example programs also use `click` and `psycopg2-binary`

# Enable pgvector

Enable the `pgvector` extension:

```
CREATE EXTENSION vector;
```

This only works if the `pgvector` extension is installed.

It may already be available, as in Aiven for PostgreSQL®

# Create our database table

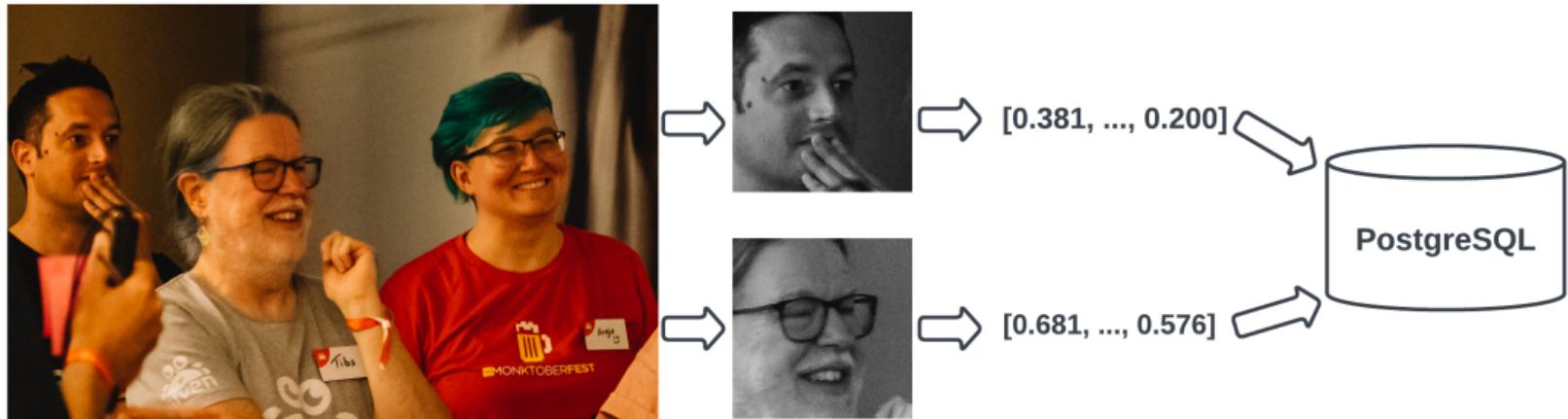
```
CREATE TABLE pictures (face text PRIMARY KEY, filename text, embedding vector(768));
```

face is the string we use to identify this particular face

filename is the name of the file we found the face in

embedding is the vector itself

# Program 1. Find faces and store their embeddings



*tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a*

# Find faces and store their embeddings

`find_faces_store_embeddings.py`

```
Usage: find_faces_store_embeddings.py [OPTIONS] IMAGE_FILES...
```

Options:

```
-p, --pg-uri TEXT  the URI for the PostgreSQL service, defaulting to  
                   $PG_SERVICE_URI if that is set  
--help              Show this message and exit.
```

*tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a*

# Find faces and store their embeddings (1)

```
def main(image_files: tuple[str], pg_uri: str):
    haar_cascade = load_algorithm()
    ibed = imgbeddings()

    for image_file in image_files:
        with psycopg2.connect(pg_uri) as conn:
            orig_image = cv2.imread(image_file, cv2.IMREAD_GRAYSCALE)
            faces = find_faces(orig_image, haar_cascade)

            write_faces_to_pg(faces, orig_image, picture_file, conn, ibed)
```

cv2 is the OpenCV package

# Find faces and store their embeddings (2)

```
def load_algorithm():
    algorithm = "haarcascade_frontalface_default.xml"
    haar_cascade = cv2.CascadeClassifier(algorithm)
    if haar_cascade.empty():
        raise GiveUp(f'Error reading algorithm file {algorithm} - no algorithm found')
    return haar_cascade
```

# Find faces and store their embeddings (3)

```
# Read the image in, as greyscale  
orig_image = cv2.imread(image_file, cv2.IMREAD_GRAYSCALE)
```

# Find faces and store their embeddings (4)

```
def find_faces(gray_image, haar_cascade):
    return haar_cascade.detectMultiScale(
        gray_image,
        scaleFactor=1.05,
        minNeighbors=2,
        minSize=(250, 250),
        #minSize=(100, 100),
    )
```

# Find faces and store their embeddings (5)

```
def write_faces_to_pg(faces, orig_image, picture_file, conn, ibed):
    file_path = Path(picture_file)
    file_base = file_path.stem
    file_posix = file_path.as_posix()

    for x, y, w, h in faces:
        # Convert to a Pillow image since that's what imgembeddings wants
        cropped_image = Image.fromarray(orig_image[y: y + h, x: x + w])
        embedding = ibed.to_embeddings(cropped_image)[0]
        face_key = f'{file_base}-{x}-{y}-{w}-{h}'

        write_to_pg(conn, face_key, file_posix, embedding)
```

# Find faces and store their embeddings (6)

And here's where we actually write to PostgreSQL

```
def write_to_pg(conn, face_key, file_name, embedding):
    with conn.cursor() as cur:
        cur.execute('INSERT INTO pictures (face_key, filename, embedding)'
                    ' VALUES (%s,%s,%s)'
                    ' ON CONFLICT (face_key) DO UPDATE'
                    '     SET filename = EXCLUDED.filename,'
                    '         embedding = EXCLUDED.embedding'
                    ' ;',
                    (face_key, file_name, embedding.tolist()))
    )
```

# Find faces and store their embeddings (7)

ON CONFLICT is interesting:

```
ON CONFLICT (face_key) DO UPDATE
  SET filename = EXCLUDED.filename,
      embedding = EXCLUDED.embedding;
```

# Program 2. Find "nearby" faces



*tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a*

# Find "nearby" faces

`find_nearby_faces.py`

```
Usage: find_nearby_faces.py [OPTIONS] FACE_FILE
```

Options:

<code>-n, --number-matches INTEGER</code>	
<code>-p, --pg-uri TEXT</code>	the URI for the PostgreSQL service, defaulting to <code>\$PG_SERVICE_URI</code> if that is set
<code>--help</code>	Show this message and exit.

*tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a*

# Find "nearby" faces (1)

```
def main(face_file: tuple[str], number_matches: int, pg_uri: str):
    haar_cascade = load_algorithm()
    ibed = imgbeddings()

    # Calculate the embedding for the face file - we assume only one face
    embedding = calc_reference_embedding(face_file, haar_cascade, ibed)

    # Convert to something that will work in SQL
    vector_str = ", ".join(str(x) for x in embedding.tolist())
    vector_str = f'[{vector_str}]'

    ask_pg_and_report(pg_uri, vector_str, number_matches)
```

tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a

# Find "nearby" faces (2)

```
def calc_reference_embedding(face_file, haar_cascade, ibed):
    orig_image = cv2.imread(picture_file, cv2.IMREAD_GRAYSCALE)
    faces = find_faces(orig_image, haar_cascade)

    # We hope there's only one face!
    cropped_images = []
    for x, y, w, h in faces:
        cropped_images.append(orig_image[y : y + h, x : x + w])

    face = Image.fromarray(cropped_images[0])
    return ibed.to_embeddings(face)[0]
```

# Find "nearby" faces (3)

In fact, in the real code it doesn't say:

```
# We hope there's only one face!
```

I couldn't resist an actual check:

```
if len(faces) == 0:  
    raise GiveUp(f"Didn't find any faces in {face_file}")  
elif len(faces) > 1:  
    raise GiveUp(f"Found more than one face in {face_file}")
```

# Find "nearby" faces (4)

Our embedding needs turning into something that SQL will understand:

```
vector_str = ", ".join(str(x) for x in embedding.tolist())
vector_str = f'[{vector_str}]'
```

# Find "nearby" faces (5)

```
def ask_pg_and_report(pg_uri, vector_str, number_matches):
    with psycopg2.connect(pg_uri) as conn:
        with conn.cursor() as cur:
            cur.execute(
                "SELECT filename FROM pictures ORDER BY embedding <-> %s LIMIT %s;",
                (vector_str, number_matches)
            )
            rows = cur.fetchall()
    print(f'Number of results: {len(rows)}')
    for index, row in enumerate(rows):
        print(f' {index}: {row[0]}'
```

# But how good is it?

779 files, 5006 faces

- 21 minutes to calculate and store the embeddings
- 3 seconds to find the 10 nearest faces

# Wednesday at Crab Week

There were 779 photos, and 5006 faces.

Going through them manually, I found 25 that had my face visible,

- some were in a crowd or obscured,
- three were of my back (!)
- two were with a false moustache

# Results the program found

And here are the first 10 matches from the program (9 are me)

```
AIVEN2752.jpg -- just me
AIVEN2839.jpg -- just me
AIVEN2838.jpg -- just me
AIVEN2806.jpg -- me in front of audience
AIVEN2808.jpg -- just me, from side
AIVEN2750.jpg -- me plus another
AIVEN2751.jpg -- me plus others
AIVEN2748.jpg -- me plus others
AIVEN2681.jpg -- me in group sitting
AIVEN3104.jpg -- not me, beard and glasses
```

# The first: AIVEN2752



*tony.ibbs@aiven.io / https://aiven.io/tibs / @much\_of\_a*

# Me in a group



*tony.ibbs@aiven.io / https://aiven.io/tibs / @much\_of\_a*

# Thursday at Crab Week

There were 574 photos and 3486 faces.

Going through them manually, I found 7 that had my face visible

- although in 4 of them I had dark glasses

# Results the program found

And here are the first 10 matches from the program (3 are me)

```
AIVEN3933.jpg -- me in audience looking down, slightly sideways
AIVEN3697.jpg -- me in group
AIVEN3670.jpg -- not me, but sort of understandable - beard & glasses
AIVEN3760.jpg -- not me, but sort of understandable - beard & glasses
AIVEN3671.jpg -- not me, but sort of understandable - beard & glasses
AIVEN3739.jpg -- me in group as in the tutorial
AIVEN3673.jpg -- not me, but sort of understandable - beard & glasses
AIVEN3999.jpg -- not me, but sort of understandable - beard & glasses
AIVEN4316.jpg -- not me, but sort of understandable - beard & (dark) glasses
AIVEN3679.jpg -- not me, but sort of understandable - beard & glasses
```

# The first: AIVEN3933



*tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a*

# As in the tutorial: AIVEN3739 (cropped)



*tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a*

# So was this a success, so far?

Definitely yes.

I learnt a lot.

I got not awful (!) results with really very low effort.

I know what to do for the next set of investigations

# What I'd do next

Improve `find_faces_store_embeddings.py`:

- Add a switch to allow setting the "face detecting" parameters
- Make a different table for each set of parameters
- Add a switch for "generate reference face"

Improve `find_nearby_faces.py`

- Add a switch to specify which face (from the db) to look for
- Add a switch to specify which table to search

# Part the fourth: Why PostgreSQL?



*tony.ibbs@aiven.io / <https://aiven.io/tibs> / @much\_of\_a*

# Why is PostgreSQL a surprising choice?

We expect Python to be a good fit for exploring ML

But PostgreSQL isn't a dedicated vector database

# So why PostgreSQL?



and/or



Images from <https://pixabay.com/>, by [OpenClipart-Vectors](#)

# It's significantly better than nothing

(faint praise indeed)

There comes a point when you need to store your embeddings in some sort of database

PostgreSQL is a *good* place to start

# We already have it

Quite often, we're already running PostgreSQL

# It can SQL all the things

This can be *really useful*:

Find me things like <this order>, that are in stock

Find the pictures of me taken in Portugal, between <these dates>

Find things that match <these qualities>, and choose the one most like <this other thing>

# PostgreSQL optimisation techniques work

You can use all the techniques you normally use in PG to optimise the query  
and can do ANALYZE on the query, too

# Indexing

Speeds up the *use* of embeddings.

- IVFFlat - exact nearest neighbours, slower
- HNSW - approximate nearest neighbours, faster

HNSW was just added in [pgvector 0.5.0](#)

# A recurring pattern

We should recognise this pattern:

Work in PostgreSQL until it's not suitable,

and *then* move to something else

# When not to use PG?

When it can't cope

When it doesn't actually do what you want

# When vectors are too big

The [pgvector Reference](#) section says:

Each vector takes `4 * dimensions + 8` bytes of storage. Each element is a single precision floating-point number (like the `real` type in Postgres), and all elements must be finite (no `NaN`, `Infinity` or `-Infinity`).

Vectors can have up to 16,000 dimensions.

# When vectors are too big to index

According to the pgvector FAQ

You can't currently **index** a vector if it has more than 2,000 dimensions

# When there are too many vectors

According to the pgvector FAQ

A non-partitioned table has a limit of 32 TB by default in Postgres. A partitioned table can have thousands of partitions of that size.

# When you need more speed

pgvector is ultimately limited by being based on a relational database that is not, itself, optimised for this task.

Remember to profile!

# When you need a missing distance function

Although this can change...

# When the queries aren't SQL

Relational databases and SQL aren't always the best solution.

For instance, OpenSearch also has vector support.

# Other tools

Is pgvector the only PostgreSQL solution?

Neon provides [pg\\_embedding](#), which uses an HNSW index

There's [an article by them](#) comparing its performance with pgvector HNSW

# A quick and not very rigorous search

There are lots of vector databases! Here are some open source solutions:

- Weaviate <https://weaviate.io/>
- Milvus <https://milvus.io/>
- Qdrant <https://qdrant.tech/>
- Vespa <https://vespa.ai/>
- Chroma <https://www.trychroma.com/>

# And some more

- OpenSearch has vector database functionality
- SingleStore vector db  
<https://www.singlestore.com/built-in-vector-database/>
- Relevance AI vector db <https://relevanceai.com/vector-db>
- The FAISS library <https://faiss.ai/>

And see lists like <https://byby.dev/vector-databases>

# The future is bright (judging from history)

Vectors are the new JSON in PostgreSQL by [Jonathan Katz](#)

Things will get better and faster and support larger vectors over the next few years.

(I'm also minded of large blob support - TOAST is always an issue, but they work on it)

# Acknowledgements

Postgres, PostgreSQL and the Slonik Logo are trademarks or registered trademarks of the PostgreSQL Community Association of Canada, and used with their permission

- ML Typewriter image from <https://unsplash.com/>, by Markus Winkler
- Penknife and Hammer images from <https://pixabay.com/>, by OpenClipart-Vectors
- Vector graph from JCC Math.Phys 191: The vector class, CC BY-SA 3.0

My colleague Francesco Tisiot for the original tutorial, and much good advice

# Fin

Get a free trial of Aiven services at  
<https://go.aiven.io/pyconuk-signup>

Also, we're hiring! See <https://aiven.io/careers>

Written in [reStructuredText](#), converted to PDF using  
[rst2pdf](#)

Slides and accompanying material  at  
<https://github.com/aiven-labs/pgvector-find-faces-talk>

