Vectorify - Complete Documentation

📋 Table of Contents

🎯 Overview

Vectorify is an enterprise-grade data transformation platform that converts various data sources (CSV, Excel, Word, SQL databases) into AI-ready formats including JSON, JSONL, and vector embeddings.

Key Benefits

- **Multi-format Input**: CSV, CSV UTF-8, Excel (.xlsx, .xls), Word (.docx, .doc), SQL databases
- **AI-Ready Output**: JSON, JSONL (JSON Lines), and vector embeddings
- **Free Embedding Models**: 3 state-of-the-art models (384D, 768D, 1024D)
- **Enterprise Security**: Secure backend processing with encrypted connections
- **Production Ready**: Built with Flask, React, and industry-standard libraries

---

✨ Features

Input Sources

Format	Extension	Support
CSV	`.csv`	✅ Full

Format	Extension	Support
CSV UTF-8	`.csv`	✅ Full
Excel	`.xlsx`, `.xls`	✅ Full
Word	`.docx`, `.doc`	✅ Full
PostgreSQL	SQL	✅ Full
MySQL	SQL	✅ Full
SQLite	SQL	✅ Full

Output Formats

Format	Extension	Use Case
JSON	`.json`	APIs, Applications, Storage
JSONL	`.jsonl`	Streaming, ML Training, Logs
Vector Embeddings	`.json`	Vector Databases, Semantic Search

Embedding Models (All Free)

Model	Dimensions	Quality	Speed	Best For
MiniLM-L6-v2	384	Good	Very Fast	Quick prototyping
MPNet-base-v2	768	Better	Fast	General purpose
BGE-large-en-v1.5	1024	Best	Moderate	Production use

💻 System Requirements

Backend (Python)

- Python 3.8 or higher
- 4GB RAM minimum (8GB recommended for embeddings)
- 2GB free disk space

Frontend (React)

- Node.js 16+ (for development)
- Modern web browser (Chrome, Firefox, Safari, Edge)

Database Support (Optional)

- PostgreSQL 12+
- MySQL 8+

- SQLite 3+

🚀 Installation

1. Clone Repository

```bash
git clone https://github.com/yourusername/vectorify.git
cd vectorify
```

2. Backend Setup

Create Virtual Environment

```bash
python -m venv venv

# On Windows
venv\Scripts\activate

# On macOS/Linux
source venv/bin/activate
```

Install Dependencies

```bash
pip install -r requirements.txt
```

3. Frontend Setup

The React frontend is included in the artifact. To integrate with backend:

```javascript
// Update API endpoint in your React app
const API_URL = 'http://localhost:5000';
```

🎬 Quick Start

1. Start Backend Server

```bash
python app.py
```

Server will start at: `http://localhost:5000`

2. Open Frontend

Open the Vectorify React app in your browser.

3. Upload Data

- **Option A**: Click "Upload File" and select CSV/Excel file
- **Option B**: Click "SQL Database" and enter connection details

4. Configure Output

- Select output formats: JSON, JSONL, and/or Embeddings
- If embeddings selected, choose a model (384D, 768D, or 1024D)

5. Generate & Download

- Click "Generate" button
- Download your outputs

📖 Usage Guide

File Upload Mode

Supported Formats

```bash
```

```
# CSV Files
data.csv
data_utf8.csv

# Excel Files
spreadsheet.xlsx
spreadsheet.xls

# Word Documents (extracts text)
document.docx
document.doc
```

Example: Upload CSV

1. Click "Upload File" tab
2. Drag & drop or browse for your CSV file
3. Wait for processing (shows record count)
4. Proceed to "Configure" tab

SQL Database Mode

Connection Parameters

python

```python
{
  "host": "localhost",      # Database host
  "port": 5432,             # Database port (PostgreSQL: 5432, MySQL: 3306)
  "database": "my_db",      # Database name
  "username": "admin",      # Database user
  "password": "secret",     # Database password
  "query": "SELECT * FROM users LIMIT 100"  # SQL query
}
```

Example: PostgreSQL Connection

sql

```sql
-- Sample Query
SELECT
  id,
  name,
  email,
  created_at
FROM users
WHERE status = 'active'
LIMIT 1000;
```

Supported Databases

- **PostgreSQL**: Port 5432 (default)
- **MySQL**: Port 3306 (default)
- **SQLite**: File-based (no host/port needed)

Output Configuration

JSON Output

json

```json
{
  "metadata": {
    "fileName": "data.csv",
    "recordCount": 1000,
    "columns": ["id", "name", "email"],
    "generatedAt": "2024-01-15T10:30:00Z",
    "format": "JSON"
  },
  "data": [
    {"id": 1, "name": "John", "email": "john@example.com"},
    {"id": 2, "name": "Jane", "email": "jane@example.com"}
  ]
}
```

JSONL Output

jsonl

```jsonl
{"id": 1, "name": "John", "email": "john@example.com"}
{"id": 2, "name": "Jane", "email": "jane@example.com"}
{"id": 3, "name": "Bob", "email": "bob@example.com"}
```

Vector Embeddings Output

json

```
{
 "vectorDatabase": {
  "dimension": 384,
  "vectorCount": 1000,
  "embeddingModel": "MiniLM-L6-v2",
  "distance": "cosine"
 },
 "vectors": [
  {
   "id": "vec_000000",
   "text": "id: 1 | name: John | email: john@example.com",
   "embedding": [0.234, -0.123, 0.456, ...],
   "metadata": {
    "rowIndex": 0,
    "recordData": {"id": 1, "name": "John"}
   }
  }
 ]
}
```

🔌 API Reference

Base URL

```
http://localhost:5000/api
```

Endpoints

1. Upload File

http

```
POST /api/upload
Content-Type: multipart/form-data

Parameters:
- file: File (CSV, Excel, Word)

Response:
{
  "success": true,
  "filename": "data.csv",
  "fileType": "CSV",
  "recordCount": 1000,
  "columns": ["id", "name", "email"],
  "filepath": "/uploads/data.csv"
}
```

2. SQL Database Connection

http

```
POST /api/sql/connect
Content-Type: application/json

Body:
{
  "dbType": "postgresql",
  "host": "localhost",
  "port": 5432,
  "database": "my_db",
  "username": "admin",
  "password": "secret",
  "query": "SELECT * FROM users LIMIT 100"
}

Response:
{
  "success": true,
  "recordCount": 100,
  "columns": ["id", "name", "email"],
  "preview": [...]
}
```

3. Convert Data

http

```
POST /api/convert
Content-Type: application/json

Body:
{
  "filepath": "/uploads/data.csv",
  "outputTypes": ["json", "jsonl", "embeddings"],
  "embeddingProvider": "sentence-transformers-mini"
}

Response:
{
  "success": true,
  "results": {
    "json": { "filepath": "...", "filename": "..." },
    "jsonl": { "filepath": "...", "filename": "..." },
    "embeddings": { "filepath": "...", "filename": "..." }
  }
}
```

4. Download Output

```http
GET /api/download/{output_type}/{filename}

Response:
File download (JSON or JSONL)
```

5. Health Check

```http
GET /api/health

Response:
{
  "status": "healthy",
  "timestamp": "2024-01-15T10:30:00Z",
  "supportedFormats": ["CSV", "XLSX", "XLS", "DOCX", "DOC", "SQL"]
}
```

6. Get Embedding Providers

```http
```

```
GET /api/providers

Response:
{
  "providers": [
    {
      "key": "sentence-transformers-mini",
      "name": "MiniLM-L6-v2",
      "dimension": 384,
      "type": "local",
      "available": true
    }
  ],
  "default": "sentence-transformers-mini"
}
```

📊 Output Formats

JSON Format

Use Cases:

- REST APIs
- Web applications
- Data storage
- Configuration files

Advantages:

- Human-readable
- Widely supported
- Rich metadata
- Nested structures

JSONL Format

Use Cases:

- Machine learning training data
- Log file processing
- Streaming pipelines

- Big data processing

Advantages:

- Memory efficient (line-by-line processing)
- Append-friendly
- Parallel processing
- OpenAI fine-tuning format

Example Usage:

python

```
# Python: Read JSONL
import json

with open('data.jsonl', 'r') as f:
    for line in f:
        record = json.loads(line)
        print(record)

# Python: Write JSONL
with open('output.jsonl', 'w') as f:
    for record in data:
        f.write(json.dumps(record) + '\n')
```

Vector Embeddings

Use Cases:

- Semantic search
- Recommendation systems
- Document similarity
- Question answering
- RAG (Retrieval Augmented Generation)

Compatible Vector Databases:

- Pinecone
- Weaviate
- Qdrant
- Milvus

- Chroma
- FAISS

Integration Examples:

Pinecone

python

```python
import pinecone
import json

# Load embeddings
with open('data_embeddings.json', 'r') as f:
    data = json.load(f)

# Initialize Pinecone
pinecone.init(api_key="your-key")
index = pinecone.Index("your-index")

# Upsert vectors
vectors = [(v['id'], v['embedding'], v['metadata'])
           for v in data['vectors']]
index.upsert(vectors=vectors)
```

Weaviate

python

```python
import weaviate
import json

client = weaviate.Client("http://localhost:8080")

with open('data_embeddings.json', 'r') as f:
    data = json.load(f)

for vector in data['vectors']:
    client.data_object.create(
        data_object={
            "text": vector['text'],
            "metadata": vector['metadata']
        },
        class_name="Document",
        vector=vector['embedding']
    )
```

Qdrant

```python
from qdrant_client import QdrantClient
from qdrant_client.models import PointStruct
import json

client = QdrantClient("localhost", port=6333)

with open('data_embeddings.json', 'r') as f:
    data = json.load(f)

points = [
    PointStruct(
        id=i,
        vector=v['embedding'],
        payload=v['metadata']
    )
    for i, v in enumerate(data['vectors'])
]

client.upsert(collection_name="my_collection", points=points)
```

🧠 Embedding Models

Model Comparison

MiniLM-L6-v2 (384D)

- **Speed**: Very Fast (~1000 records/sec)
- **Quality**: Good
- **Memory**: ~100MB
- **Best For**: Quick prototyping, testing, real-time applications
- **Use Cases**: Chatbots, quick search, prototypes

MPNet-base-v2 (768D)

- **Speed**: Fast (~500 records/sec)
- **Quality**: Better
- **Memory**: ~400MB

- **Best For**: Production applications, general purpose
- **Use Cases**: Document search, content recommendations, semantic matching

BGE-large-en-v1.5 (1024D)

- **Speed**: Moderate (~250 records/sec)
- **Quality**: Best (State-of-the-art)
- **Memory**: ~1GB
- **Best For**: High-accuracy requirements, enterprise applications
- **Use Cases**: Legal documents, medical records, high-stakes search

Choosing the Right Model

python

```python
# Decision Tree
if use_case == "quick_prototype":
    model = "MiniLM-L6-v2"  # 384D
elif use_case == "production_general":
    model = "MPNet-base-v2"  # 768D
elif use_case == "high_accuracy":
    model = "BGE-large-en-v1.5"  # 1024D
```

Adding Custom Models

To add your own embedding models:

python

```python
# In enhanced-python-backend.py

EMBEDDING_PROVIDERS['custom-model'] = {
    'name': 'your-model-name',
    'dimension': 512,
    'type': 'local',
    'model': None
}

# Add to model_map in load_local_model()
model_map['custom-model'] = 'huggingface/your-model-name'
```

🚢 Production Deployment

Docker Deployment

Dockerfile

```dockerfile
FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5000

CMD ["gunicorn", "--bind", "0.0.0.0:5000", "--workers", "4", "app:app"]
```

docker-compose.yml

```yaml
```

```
version: '3.8'

services:
  backend:
    build: .
    ports:
      - "5000:5000"
    environment:
      - FLASK_ENV=production
      - DATABASE_URL=postgresql://user:pass@db:5432/vectorify
    depends_on:
      - db
    volumes:
      - ./uploads:/app/uploads
      - ./outputs:/app/outputs

  db:
    image: postgres:14
    environment:
      POSTGRES_DB: vectorify
      POSTGRES_USER: admin
      POSTGRES_PASSWORD: secret
    volumes:
      - postgres_data:/var/lib/postgresql/data

volumes:
  postgres_data:
```

Environment Variables

Create `.env` file:

```
bash
```

```
# Flask Configuration
FLASK_ENV=production
SECRET_KEY=your-secret-key-here

# Database
DATABASE_URL=postgresql://user:pass@localhost:5432/vectorify

# Optional: API Keys for premium embeddings
OPENAI_API_KEY=sk-...
COHERE_API_KEY=...

# Storage
UPLOAD_FOLDER=./uploads
OUTPUT_FOLDER=./outputs
MAX_CONTENT_LENGTH=100000000  # 100MB
```

Production Considerations

Security

python

```
# Enable CORS with specific origins
CORS(app, origins=["https://yourdomain.com"])

# Add authentication middleware
from functools import wraps

def require_api_key(f):
    @wraps(f)
    def decorated(*args, **kwargs):
        api_key = request.headers.get('X-API-Key')
        if api_key != os.getenv('API_KEY'):
            return jsonify({'error': 'Unauthorized'}), 401
        return f(*args, **kwargs)
    return decorated

@app.route('/api/convert')
@require_api_key
def convert_data():
    # ...
```

Performance Optimization

python

```
# Use Gunicorn for production
gunicorn --workers 4 --threads 2 --timeout 300 app:app

# Add Redis caching
from flask_caching import Cache

cache = Cache(app, config={'CACHE_TYPE': 'redis'})

@cache.memoize(timeout=3600)
def generate_embeddings(text, model):
    # ...
```

Monitoring

```python
# Add logging
import logging

logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s - %(name)s - %(levelname)s - %(message)s',
    handlers=[
        logging.FileHandler('vectorify.log'),
        logging.StreamHandler()
    ]
)

# Add metrics endpoint
from prometheus_flask_exporter import PrometheusMetrics

metrics = PrometheusMetrics(app)
```

Cloud Deployment

AWS (Elastic Beanstalk)

```bash
eb init -p python-3.9 vectorify
eb create vectorify-env
eb deploy
```

Google Cloud (Cloud Run)

```bash
```

```
gcloud builds submit --tag gcr.io/PROJECT_ID/vectorify
gcloud run deploy --image gcr.io/PROJECT_ID/vectorify --platform managed
```

Azure (App Service)

```bash
az webapp up --name vectorify --runtime "PYTHON|3.9"
```

🔧 Troubleshooting

Common Issues

1. Module Not Found Error

```bash
# Solution: Ensure all dependencies are installed
pip install -r requirements.txt

# Verify installation
pip list | grep sentence-transformers
```

2. CUDA/GPU Issues

```bash
# For CPU-only usage, install CPU version of PyTorch
pip install torch --index-url https://download.pytorch.org/whl/cpu
```

3. Memory Error During Embeddings

```python
# Solution: Process in smaller batches
# Reduce batch size in the code
chunk_size = 50  # Instead of default 100
```

4. SQL Connection Failed

```bash
```

```
# Check database is running
# PostgreSQL
sudo systemctl status postgresql

# MySQL
sudo systemctl status mysql

# Test connection
psql -h localhost -U admin -d vectorify
```

5. CORS Error

```python
# Update CORS settings
CORS(app, resources={
    r"/api/*": {
        "origins": "*",  # For development only
        "methods": ["GET", "POST"],
        "allow_headers": ["Content-Type"]
    }
})
```

Debug Mode

Enable debug logging:

```python
import logging
logging.basicConfig(level=logging.DEBUG)

app.run(debug=True)
```

Performance Issues

Slow Embeddings Generation

- Use smaller model (MiniLM-L6-v2)
- Reduce batch size
- Enable GPU if available
- Consider API-based embeddings (OpenAI, Cohere)

Large File Processing

```python
# Process in chunks
chunk_size = 1000
for i in range(0, len(df), chunk_size):
    chunk = df[i:i+chunk_size]
    process_chunk(chunk)
```

📞 Support

Documentation

- GitHub: https://github.com/yourusername/vectorify
- Issues: https://github.com/yourusername/vectorify/issues

Community

- Discord: [Your Discord Link]
- Stack Overflow: Tag `vectorify`

Enterprise Support

- Email: support@vectorify.com
- Custom integrations and training available

📄 License

MIT License - See LICENSE file for details

🙏 Acknowledgments

- Sentence Transformers by UKPLab
- Flask framework
- React community
- All open-source contributors

Version: 1.0.0
Last Updated: November 2024
Status: Production Ready ✅