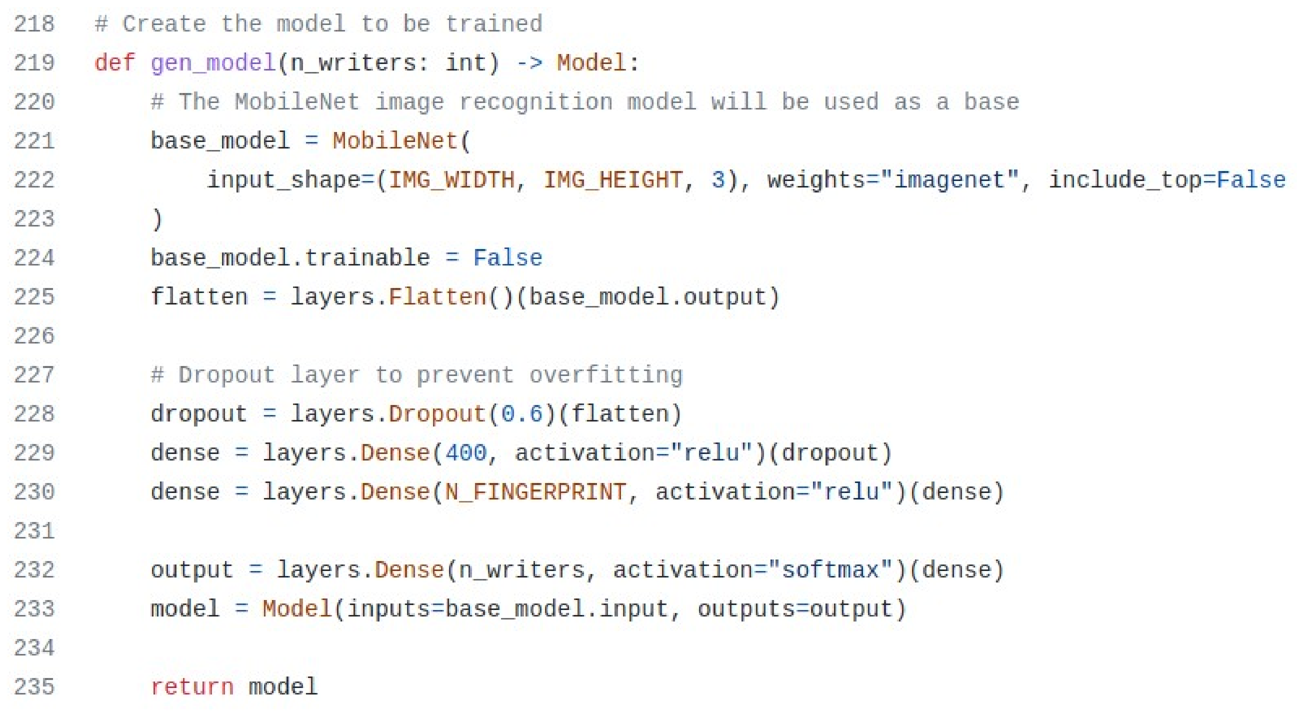
Criterion C: Development

Thomas Faulhaber

Key techniques used in the author-id-model project (the AI) include:

- Neural network

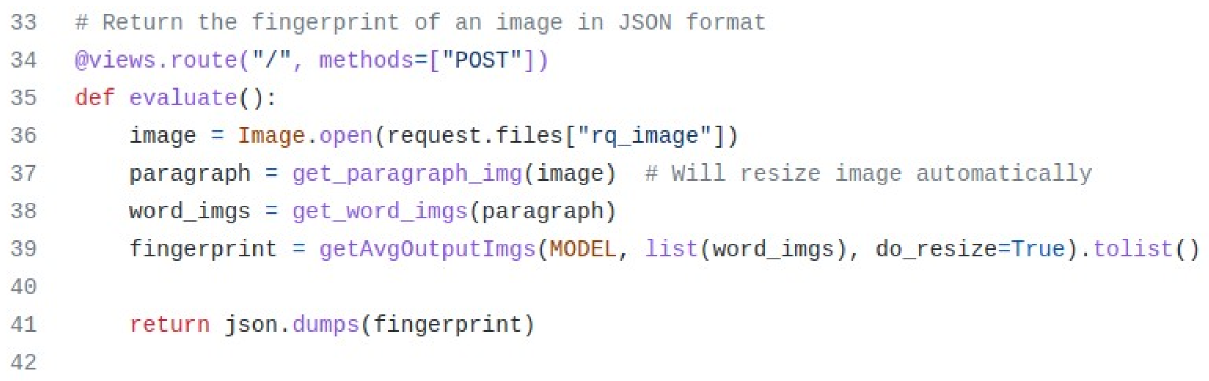


```python
# Create the model to be trained
def gen_model(n_writers: int) -> Model:
    # The MobileNet image recognition model will be used as a base
    base_model = MobileNet(
        input_shape=(IMG_WIDTH, IMG_HEIGHT, 3), weights="imagenet", include_top=False
    )
    base_model.trainable = False
    flatten = layers.Flatten()(base_model.output)

    # Dropout layer to prevent overfitting
    dropout = layers.Dropout(0.6)(flatten)
    dense = layers.Dense(400, activation="relu")(dropout)
    dense = layers.Dense(N_FINGERPRINT, activation="relu")(dense)

    output = layers.Dense(n_writers, activation="softmax")(dense)
    model = Model(inputs=base_model.input, outputs=output)

    return model
```

The backbone of the entire project is the neural network that is used for handwriting identification. The way it works is as follows:

1. A third-party tool (https://github.com/awslabs/handwritten-text-recognition-for-apache-mxnet/) is used to extract each individual word from a handwriting sample.
2. Each word is fed into the neural net as an image. It returns an array of floating-point numbers of an arbitrary, fixed length (200 numbers in the final code), representing (in theory) the different features present in the handwriting. This array is the word fingerprint.
3. A sample fingerprint is produced by taking the arithmetic mean of all the word fingerprints.
4. Ideally, the sample fingerprint for a sample by Author A should be closer in n-dimensional space (200-dimensional space in this case) to another sample by Author A than it is to any sample by any other author.

- API

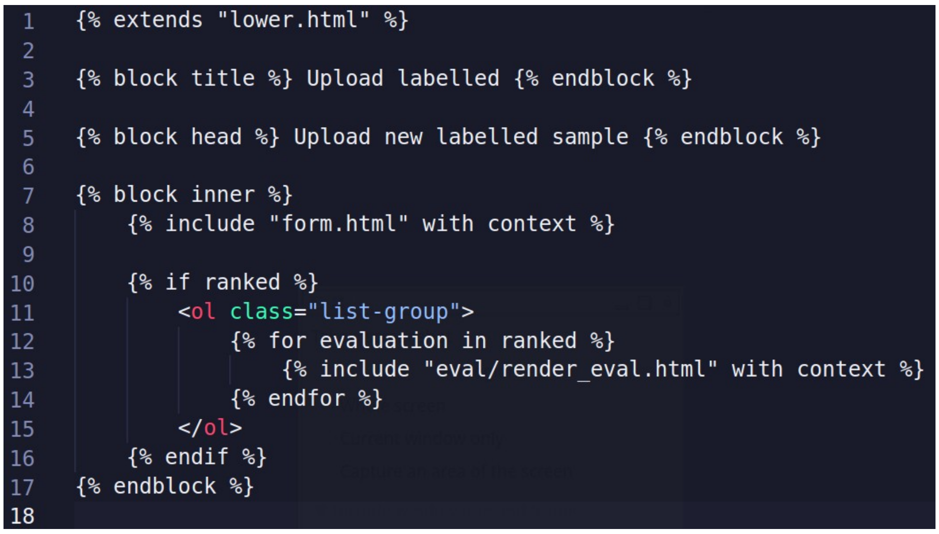


```python
# Return the fingerprint of an image in JSON format
@views.route("/", methods=["POST"])
def evaluate():
    image = Image.open(request.files["rq_image"])
    paragraph = get_paragraph_img(image)  # Will resize image automatically
    word_imgs = get_word_imgs(paragraph)
    fingerprint = getAvgOutputImgs(MODEL, list(word_imgs), do_resize=True).tolist()

    return json.dumps(fingerprint)

```

In order to keep the AI- and web-related portions of the project separate, I wrote a very rudimentary API that allows the neural net to be queried over HTTP. This way, I don’t have to worry about code interoperability between the two programs (which is a pain in Python due to its weird module system), and the project structure for both programs is much less cluttered than it would otherwise be.

Key techniques used in the author-id-server project (the web app) include:

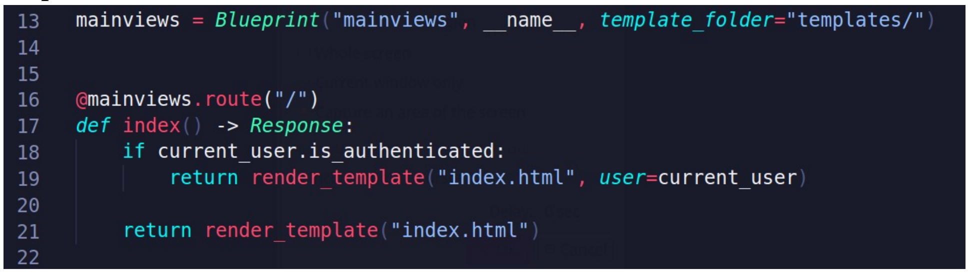
- Jinja2 templating



```
{% extends "lower.html" %}

{% block title %} Upload labelled {% endblock %}

{% block head %} Upload new labelled sample {% endblock %}

{% block inner %}
    {% include "form.html" with context %}

    {% if ranked %}
        <ol class="list-group">
            {% for evaluation in ranked %}
                {% include "eval/render_eval.html" with context %}
            {% endfor %}
        </ol>
    {% endif %}
{% endblock %}

```

The Flask library uses the Jinja2 templating language by default for server-side generation. This allows me to insert data from the backend into HTML to be served to the client. The ability to dynamically generate webpages on the backend meant that I didn’t have to write any Javascript, which was nice because I don’t like writing Javascript.

- Flask blueprints



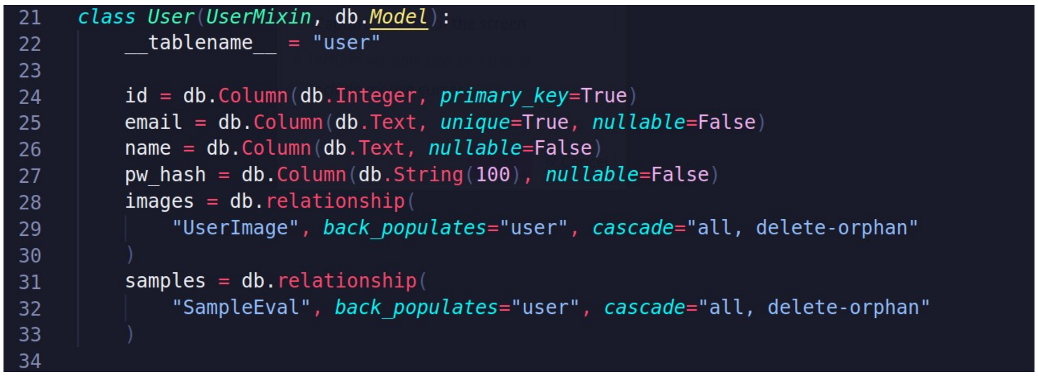
```python
mainviews = Blueprint("mainviews", __name__, template_folder="templates/")


@mainviews.route("/")
def index() -> Response:
    if current_user.is_authenticated:
        return render_template("index.html", user=current_user)

    return render_template("index.html")

```

A typical Flask app consists of a module containing a set of views, each of which is accessible on the server through an HTTP route (e.g. the “user login” view, a.k.a. the login page, may be accessible through the route “/login,” corresponding to https://yourdomainhere.com/login as a URL). In Flask, a view is a function that produces a response object which the server sends to the client.

The Flask Blueprint module allows the developer to have related views be grouped together into separate Python modules, each called a blueprint. For example, this app has a blueprint for views related to user authentication. These modules can then be imported by the main module (the one which is run upon deployment) and incorporated, all together, into one Flask app. This way, all of the routes from each of several blueprints can be served on the same server.

Because a module and a source file are basically the same thing in Python, I would need to have all of your views in the same file if I wasn’t using blueprints. Since each view is a Python function which may be rather lengthy and involved, this would lead to one very long and hard-to-navigate source file.

- SQLAlchemy ORM

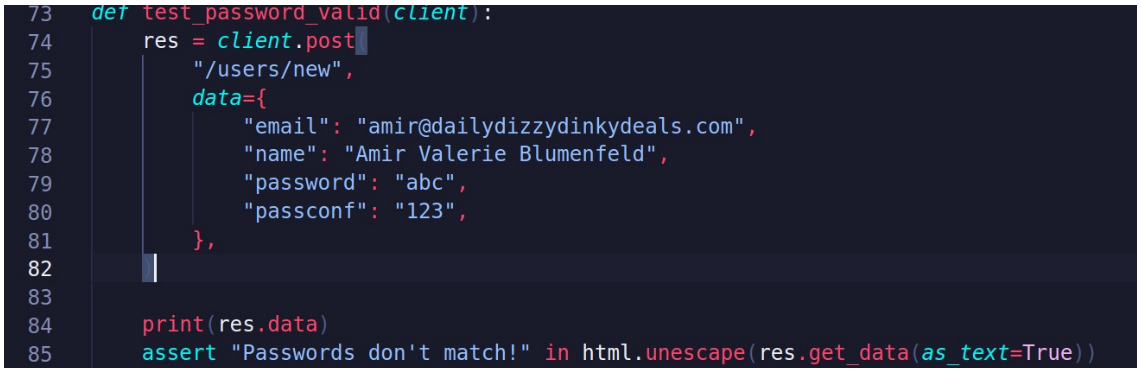


```python
class User(UserMixin, db.Model):
    __tablename__ = "user"

    id = db.Column(db.Integer, primary_key=True)
    email = db.Column(db.Text, unique=True, nullable=False)
    name = db.Column(db.Text, nullable=False)
    pw_hash = db.Column(db.String(100), nullable=False)
    images = db.relationship(
        "UserImage", back_populates="user", cascade="all, delete-orphan"
    )
    samples = db.relationship(
        "SampleEval", back_populates="user", cascade="all, delete-orphan"
    )

```

The flask-sqlalchemy PyPI package allows integration between Flask and the SQLAlchemy database manipulation library. This provides an ORM that allows data stored in the sqlite database to be accessed as Python objects. This allowed me to access the database, which contains data about the app's users and the content they've uploaded to the server, without having to type out SQL queries and also provided data sanitation.

The code snippet above shows how a database model representing a User is set up in SQLAlchemy. Data about the user is easily accessed with this class as an interface (for example, a user's email can easily be retrieved by accessing the "email" field of an instance. No SQL queries need to be written by hand).

- Unit testing



```python
def test_password_valid(client):
    res = client.post(
        "/users/new",
        data={
            "email": "amir@dailydizzydinkydeals.com",
            "name": "Amir Valerie Blumenfeld",
            "password": "abc",
            "passconf": "123",
        },
    )

    print(res.data)
    assert "Passwords don't match!" in html.unescape(res.get_data(as_text=True))
```

Writing tests ahead of type using the pytest package made it so that I didn't have to check that I hadn't broken anything manually every time I added something.

The code snippet above shows a simple test in which I'm checking to make sure that trying to create a user fails if the password and password confirmation fields don’t contain matching data. I do this by querying a “mock” client with invalid data and then checking to make sure that the response contains the correct error.