Introduction to FLASK for Model Deployment:

Step-by-Step Guide for Database Integration

Use Case: Text Sentiment Analysis

<u>Prerequisites:</u>

1) **Python 3.7 and above with pip installed**
2) **Access to a code editor (e.g. Visual Studio Code & Notepad++)**
   Note: The following guide will be using Visual Studio Code.
3) **Access to terminal console**
   Note: The following guide will be mainly using Ubuntu 20.04.

Contents Page

A. Installation of Database & ORM Wrappers

A1. PostgreSQL, Flask & SQLAlchemy

PostgreSQL is an open-source object-relational database system that is known for reliability, scalability, and performance. Other commonly used relational databases include MySQL and SQLite. As Flask does not support databases natively, the Flask-SQLAlchemy is a useful extension that provides a Flask-friendly wrapper to the popular SQLAlchemy package, which is an Object Relational Mapper (ORM).

An ORM is a code library that automates the transfer of data stored in relational databases tables into objects that are more commonly used in application code. With ORM, you will be easily able to associate user-defined Python classes with database tables, thereby allowing you to express database queries in terms of the user defined classes and their defined relationships between each other. For more details on SQLAlchemy, please refer to the following link: SQLAlchemy.

A2. Steps to Install PostgreSQL and SQLAlchemy

For the purpose of this guide, we will be setting up a PostgreSQL database locally and using manual Object Relational Mapping. If you are comfortable with ORM wrappers and would like to connect your existing database using SQLAlchemy, you can refer to the following links: Flask-SQLAlchemy documentation & Stack Overflow Link

Note: The following commands should be executed in your terminal (and not typed in a Python file), and not within the virtual environment.

1. Install PostgreSQL on your machine.

```
>> sudo apt-get install postgresql postgresql-contrib
```

2. To configure PostgreSQL, we will need to: (i) create a database; (ii) create a user; and (iii) grant the user access to the database that has been created.

a. Start the PostgreSQL database server. You will need to do this step whenever you reboot your machine, if you have not configured the database server to start automatically.

```
>> sudo service postgresql start
```

b. [OPTIONAL] If you decide that you would like to configure the database server to start automatically whenever your machine is booted up, you can run the following command. This eliminates the need to do Step 1 (to start the PostgreSQL database server) every time that your machine is booted up.

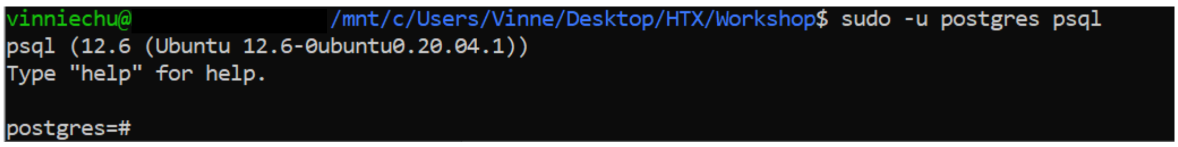
For Mac users, please refer to the following link for a similar command using launchd: link

```
>> sudo systemctl enable postgresql
```

c. Connect to psql (interactive command line terminal interface for working with PostgreSQL) using the following command.

```
>> sudo -u postgres psql
```

If your installation above is successful, you should enter the PostgreSQL environment (where you can execute SQL statements), which is similar to the one below:



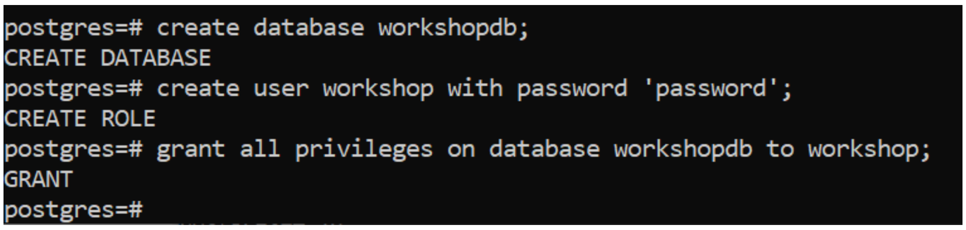
```
vinniechu@[redacted]  /mnt/c/Users/Vinne/Desktop/HTX/Workshop$ sudo -u postgres psql
psql (12.6 (Ubuntu 12.6-0ubuntu0.20.04.1))
Type "help" for help.

postgres=#
```

3. Type in the following three commands in psql to: (i) create a database; (ii) create a user; and (iii) grant user access to the database that has been created.

Note: You should replace [database_name], [username] and [some_password] with the database name, user name and password of your choice, respectively.

```
>> create database [database_name];
>> create user [username] with password '[some_password]';
>> grant all privileges on database [database_name] to [username];
```

Figure below shows how the commands look like when typed in the terminal, with:
[database_name] as workshopdb
[username] as workshop
[some_password] as password



```
postgres=# create database workshopdb;
CREATE DATABASE
postgres=# create user workshop with password 'password';
CREATE ROLE
postgres=# grant all privileges on database workshopdb to workshop;
GRANT
postgres=#
```

5. Quit the psql terminal by typing \q.

Note: Ensure that you have exited from the psql terminal before proceeding with the following sections. The installation and configuration of PostgreSQL only needs to be done once, for each project on each machine.

6. If you have set up your virtual environment and installed the packages via requirements.txt in the guide "1) Intro to FLASK model deployment - Guide for API Creation", you can skip this step. Otherwise please install the following packages:

```
>> pip install flask==1.1.4 flask_sqlalchemy flask_script flask_migrate==2.5.3
psycopg2-binary
```

B. Data Modelling with ORM

Based on the main use case of sentiment text analysis for the guides, we will be setting up 2 tables named *Lexicon_dictionary* and *Lentiment_score* in our database to store custom sentiment scores for each lexicon, which is one of the common methods used in lexicon-based text sentiment analysis. This is how the tables might look like:

Lexicon_dictionary

lexiconID	lexicon	language
1	happy	english
2	sad	english
3	excited	english

sentiment_score

scoreID	sentiment	score	lexiconID
1	positive	0.99	1
2	positive	-0.99	2
3	negative	-0.80	1

Note: The lexiconID in the Sentiment_score is the foreign key for the `Lexicon_dictionary` table.

We can represent the above `lexicon_dictionary` in the file *models.py* using the class below:

```
from app import db

class Lexicon_dictionary(db.Model):
    __tablename__ = 'lexicon_dictionary'

    lexiconID = db.Column(db.Integer, primary_key=True)
    lexicon = db.Column(db.String(1000), unique=False, nullable=False)
    language = db.Column(db.String(1000), unique=False, nullable=False)

    #one-to-many model
    pos_neg_scores = db.relationship('Sentiment_score', back_populates =
'lexicons_pos_neg', cascade ='all', lazy=True, uselist=True)

    def __init__(self, lexicon, language):
        self.lexicon = lexicon
        self.language = language

    def serialize(self):
        result = {'lexicon':self.lexicon, 'language':self.language}
        return result
```

- As each lexicon can have many sentiment scores, the *pos_neg_scores* variable sets the one-to-many relationship with the *Sentiment_score* table.
- It is often insufficient to specify only the table name and the fields in the data model. You should include the following two methods when specifying the data model: (i) __init__, which provides initialization of a object with some initial values; (ii) serialize, which is the process of converting structured data into a format (e.g., JSON) that allows for sharing/storage of data in a form that allows recovery of its original structure.

Similarly, we can also represent the above `Sentiment_score` in the file *models.py* using the class below:

```
from app import db

class Sentiment_score(db.Model):
    __tablename__ = 'sentiment_score'

    scoreID = db.Column(db.Integer, primary_key=True)
    sentiment = db.Column(db.String(1000), unique=False, nullable=False)
    score = db.Column(db.Float(), unique=False, nullable=False)
    lexiconID = db.Column(db.Integer, db.ForeignKey('lexicon_dictionary.lexiconID'),
unique=False, nullable=False)

    #one-to-many model
    lexicons_pos_neg = db.relationship('Lexicon_dictionary', back_populates =
'pos_neg_scores')

    def __init__(self, score, lexiconID, sentiment):
        self.score = score
        self.sentiment = sentiment
        self.lexiconID = lexiconID

    def serialize(self):
        result = {'lexicon':self.lexiconID, 'score':self.score, 'sentiment':
self.sentiment}
        return result
```

- The *`Lexicons_pos_neg`* variable completes the one-to-many relationship with the *`Lexicon_dictionary`* table by stating which variable to back populate to.

You can now save and close the file *models.py*.

C. Database Migration Using Flask-Migrate

1. To bind Flask-migrate to your Flask app, add the following code snipper in the file *manage.py*:

```
from flask_script import Manager, Server
from flask_migrate import Migrate, MigrateCommand

from app import app, db

migrate = Migrate(app, db)
manager = Manager(app)

manager.add_command('runserver', Server(host='localhost'))

manager.add_command('db', MigrateCommand)

if __name__ == '__main__':
    manager.run()
```

You can now save and close the file manage.py.

2. For the main Flask app (*i.e. main.py)*, import the necessary libraries/data models at the beginning of the file and configure the connection to the database:

```
from flask import Flask, jsonify, request, render_template, json

# import the necessary libraries for database access/functions
from flask_sqlalchemy import SQLAlchemy
from sqlalchemy.sql.functions import func
from sqlalchemy import exc

app = Flask(__name__)

# configure the connection to the database
app.config['SQLALCHEMY_DATABASE_URI'] =
'postgresql://workshop:password@localhost:5432/workshopdb'
app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = False
db = SQLAlchemy(app)

from utils import vader_model, bert_model

# import the data models
from models import Lexicon_dictionary, Sentiment_score
```

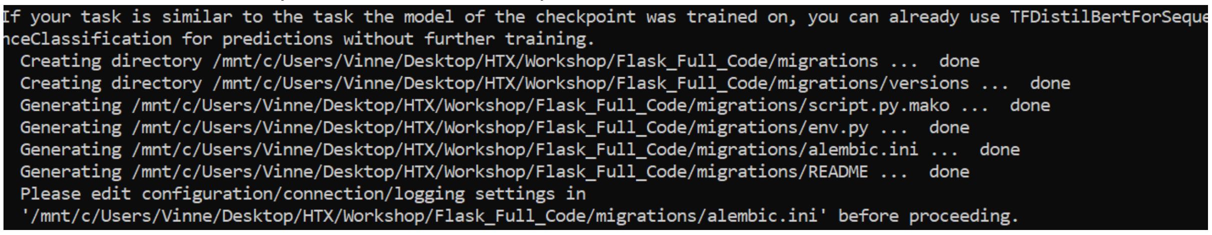
- To configure the Flask application's connection to the PostgreSQL database using SQLAlchemy, the app.config['SQLALCHEMY_DATABASE_URI'] must be initialised to the following value: **postgresql://[username]:[password]@[hostname]:[port]/[database_name]**. Since the database will be set up locally (i.e., on the same machine), the [hostname] can be "localhost" while [port] is "5432" (i.e., PostgreSQL default port).
- The data models must also be imported from *models.py* as seen from the last line in the code snippet - "from models import Lexicon_dictionary, Sentiment_score".

With the above files, we are now ready to handle database migrations in Flask.

3. Create a migration repository using:

```
>> cd Workshop
>> python3 python manage.py db init
```

If the init is successful, you should see an output like the one below:

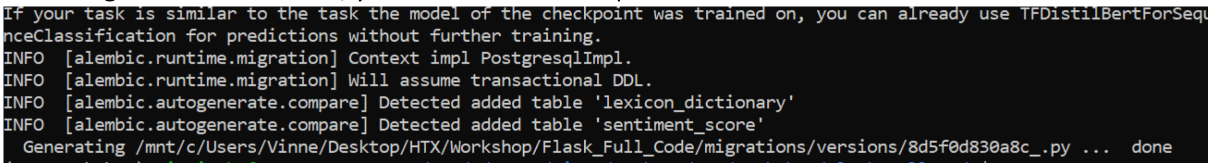


```
If your task is similar to the task the model of the checkpoint was trained on, you can already use TFDistilBertForSeque
nceClassification for predictions without further training.
  Creating directory /mnt/c/Users/Vinne/Desktop/HTX/Workshop/Flask_Full_Code/migrations ...  done
  Creating directory /mnt/c/Users/Vinne/Desktop/HTX/Workshop/Flask_Full_Code/migrations/versions ...  done
  Generating /mnt/c/Users/Vinne/Desktop/HTX/Workshop/Flask_Full_Code/migrations/script.py.mako ...  done
  Generating /mnt/c/Users/Vinne/Desktop/HTX/Workshop/Flask_Full_Code/migrations/env.py ...  done
  Generating /mnt/c/Users/Vinne/Desktop/HTX/Workshop/Flask_Full_Code/migrations/alembic.ini ...  done
  Generating /mnt/c/Users/Vinne/Desktop/HTX/Workshop/Flask_Full_Code/migrations/README ...  done
  Please edit configuration/connection/logging settings in
  '/mnt/c/Users/Vinne/Desktop/HTX/Workshop/Flask_Full_Code/migrations/alembic.ini' before proceeding.
```

4. You can then generate an initial migration using the following command:

```
>> python3 python manage.py db migrate
```

If the migration is successful, you should see an output like the one below:

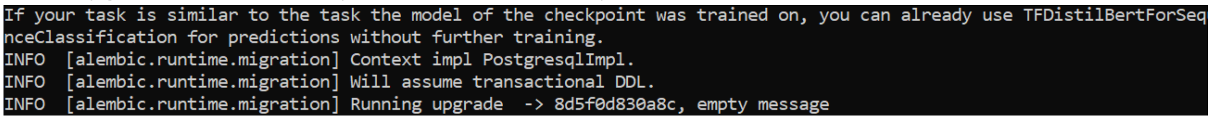


```
If your task is similar to the task the model of the checkpoint was trained on, you can already use TFDistilBertForSequ
nceClassification for predictions without further training.
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.autogenerate.compare] Detected added table 'lexicon_dictionary'
INFO  [alembic.autogenerate.compare] Detected added table 'sentiment_score'
  Generating /mnt/c/Users/Vinne/Desktop/HTX/Workshop/Flask_Full_Code/migrations/versions/8d5f0d830a8c_.py ...  done
```

5. You can now apply the migration changes to the database.

```
>> python3 python manage.py db upgrade
```

If the upgrade is successful, you should see an output like the one below:

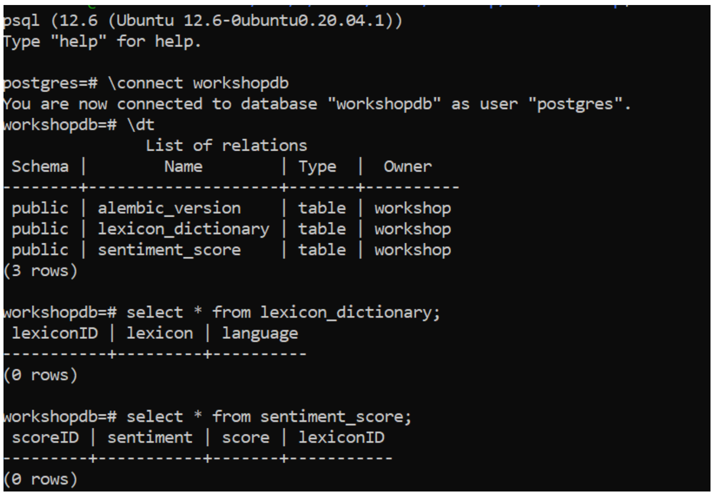


```
If your task is similar to the task the model of the checkpoint was trained on, you can already use TFDistilBertForSeq
nceClassification for predictions without further training.
INFO  [alembic.runtime.migration] Context impl PostgresqlImpl.
INFO  [alembic.runtime.migration] Will assume transactional DDL.
INFO  [alembic.runtime.migration] Running upgrade  -> 8d5f0d830a8c, empty message
```

6. To check if your database is initialised successfully, you can open a new terminal window and enter the PostgreSQL environment. For a recap on how to do so, you can refer to Section A, Step 2c. Run the following command to connect to the database:

```
>> \connect workshopdb
```

When running the *"SELECT *"* SQL statements, you should see an output of empty tables, similar to the screenshot below:



```
psql (12.6 (Ubuntu 12.6-0ubuntu0.20.04.1))
Type "help" for help.

postgres=# \connect workshopdb
You are now connected to database "workshopdb" as user "postgres".
workshopdb=# \dt
              List of relations
 Schema |        Name        | Type  |  Owner
--------+--------------------+-------+----------
 public | alembic_version    | table | workshop
 public | lexicon_dictionary | table | workshop
 public | sentiment_score    | table | workshop
(3 rows)

workshopdb=# select * from lexicon_dictionary;
 lexiconID | lexicon | language
-----------+---------+----------
(0 rows)

workshopdb=# select * from sentiment_score;
 scoreID | sentiment | score | lexiconID
---------+-----------+-------+-----------
(0 rows)
```

D. Setting Up Routes (CRUD)

Recall that the CRUD operations for persistent storage are as follows:

Operation	SQL	RESTFul
Create	INSERT	POST
Read	SELECT	GET
Update	UPDATE	PUT
Delete	DELETE	DELETE

D1. POST Request (For insertion of new data rows)

The POST operation allows the user to send a request to create an object, which is then added to the database. For instance, in the case of our use case, the following code snippet adds a new lexicon to our *Lexicon_dictionary* table:

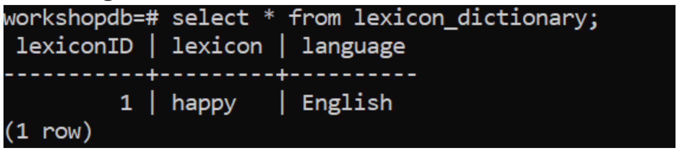
```
@app.route('/add_lexicon/', methods=['POST'])
def add_lexicon():
    try:
        lexicon = request.json['lexicon']
        language = request.json['language']
        new_lexicon = Lexicon_dictionary(lexicon=lexicon, language=language)
        db.session.add(new_lexicon)
        db.session.commit()
        return jsonify('{} was created'.format(new_lexicon))
    except exc.IntegrityError as e:
        return "This lexicon already exists."
    except KeyError as e:
        return 'Please fill in all manadatory details in the following format:
{"lexicon": "happy", "language":"english"}'
    except Exception as e:
        return(str(e))
```

- To add the new lexicon into the database, the following line is used: *db.session.add(new_lexicon)*
- Most importantly, to commit the changes back to the database, the following line must be added: *db.session.commit().*
- Exceptions are caught to make the errors more interpretable and user-friendly.

To test the above code, you can run the following code snippet found in the notebook “Accessing API endpoints.ipynb” in the Notebooks folder.

```
# to add lexicon
url = 'http://localhost:5000/add_lexicon/'
data = {'language': 'English', 'lexicon': 'happy'}

x = requests.post(url, json=data)
print(x) # if the request is successful, this should print “<Response [200]>”
print(x.json()) # if the addition is successful, this should print '<Lexicon_dictionary
n> was created'
```

Other than the above printed successful messages, you can open a new terminal window and enter the PostgreSQL environment. You should see the following added rows:



```
workshopdb=# select * from lexicon_dictionary;
 lexiconID | lexicon | language
-----------+---------+----------
         1 | happy   | English
(1 row)
```

To add the sentiment score which had a one-to-many relationship with the *Lexicon_dictionary* table, you can add the following code snippet:

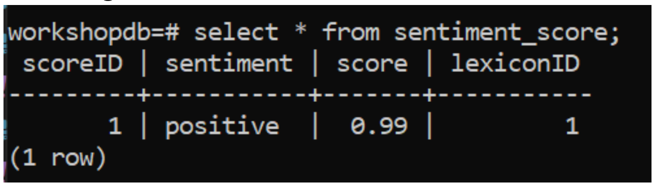
```
@app.route('/add_sentiment_score/', methods=['POST'])
def add_sentiment_score():
    try:
        lexicon = request.json['lexicon']
        language = request.json['language']
        score = request.json['score']
        sentiment = request.json['sentiment']

        # check if the lexicon is present in lexicon_dictionary table
        check = Lexicon_dictionary.query.filter_by(lexicon=lexicon,
language=language).first()
        if check == None:
            return "This lexicon does not exist!"
        else:
            lexiconID = check.lexiconID

        new_sentiment_score = Sentiment_score(lexiconID=lexiconID, score=score,
sentiment=sentiment)
        db.session.add(new_sentiment_score)
        db.session.commit()
        return jsonify('{} was created'.format(new_sentiment_score))
    except KeyError as e:
        return 'Please fill in all manadatory details in the following format:
{"lexicon":"happy", "language":"english", "score": 0.9, "sentiment": "positive"}'
    except Exception as e:
        return(str(e))
```

- To ensure our database is normalised, a lexiconID is used as a foreign key to the *Lexicon_dictionary* table. As such, when the user inputs the language and lexicon, the code first queries and retrieves the lexiconID from the *Lexicon_dictionary* table.
- After retrieving the lexiconID, the new data row is then similarly added and committed to the *Sentiment_score* table.

To test the above code, you can run the following code snippet found in the notebook "Accessing API endpoints.ipynb" in the Notebooks folder.

```
# to add sentiment score for the lexicon
url = 'http://localhost:5000/add_sentiment_score/'
data = {'language': 'English', 'lexicon': 'happy', 'sentiment': 'positive', 'score':
0.99}

x = requests.post(url, json=data)
print(x) # if the request is successful, this should print "<Response [200]>"
print(x.json()) # if the addition is successful, this should print ' <Sentiment_score n>
was created'
```

Other than the above printed successful messages, you can open a new terminal window and enter the PostgreSQL environment. You should see the following added rows:



```
workshopdb=# select * from sentiment_score;
 scoreID | sentiment | score | lexiconID
---------+-----------+-------+-----------
       1 | positive  |  0.99 |         1
(1 row)
```

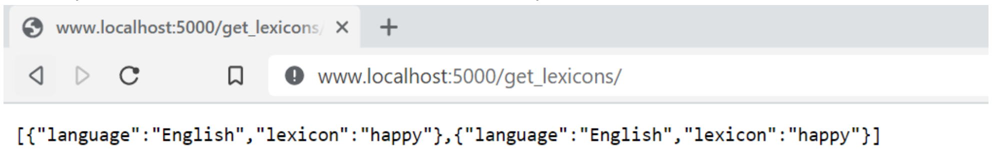
D2. GET Request (For retrieval data rows)

To retrieve the rows from a table, the following code snippet can be added:

```
@app.route('/get_lexicons/', methods=['GET'])
def get_lexicons():
       lexicons = Lexicon_dictionary.query.all()
       return jsonify([c.serialize() for c in lexicons])
```

Now its your turn to get your hands dirty! Try writing the code to access this API endpoint using the Python request package. You can refer to the example from the guide "1) Intro to FLASK model deployment - Guide for API Creation" for hints!

Other than using the Python request package, you can also access the route via your web browser for a quick examination of the results. A sample is as shown below:


www.localhost:5000/get_lexicons/
www.localhost:5000/get_lexicons/

```
[{"language":"English","lexicon":"happy"},{"language":"English","lexicon":"happy"}]
```

E. References & Extra Readings

1. More about Object-relational Mappers (ORM)
2. Declaring Models
3. Connecting Flask to a Database with Flask SQLAlchemy
4. More examples on using SQLAlchemy with Flask and PostgreSQL

--THE END---