(Copy from here)
Custom Classifier Design Doc

Authors: zacharias.fisches@, vasily.vitchevsky@

Description

This document describes a service where a user can train and run a classifier on their data. By *user* we mean a data analyst who can write SQL queries, but doesn't have access to ML tools such as linear regression.

Eventually, a user will provide input data (e.g. SQL query result), choose a classifier (e.g. linear regression), train it, and finally use the classifier on yet another input data (e.g. SQL query result). The result will be saved to a new table.
As the input data is updated roughly every week, we have to presumably enable automatic weekly re-runs of the training of the model.

Overview

Frontend

The website is where the user works and submits train/predict requests.
The website contains following pages:

1. Page 1: List of all models, their status. Have train / predict buttons. Later - stop button.
2. Page 2: Create / Modify a model.
    a. Model Name
    b. Description
    c. Train SQL query for training data
    d. Dropdown: classifier selection
    e. Eval SQL query for evaluation data

Backend

Train/predict requests from the frontend are sent to a backend (right now: the web server). The backend sends long running jobs to workers.
The result of a *training* job is a trained class instance. We save (pickle) the result to a models DB, along with the model parameters.
The result of a *predict* job is the predictions. We save them to a new table in the DB.

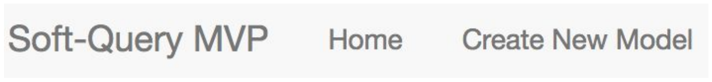
Details

Frontend

One solution would be to integrate with *redash*, but we then may have to modify our code when a new redash version is released, and it's probably more difficult. Therefore, for the first version we have a custom website.

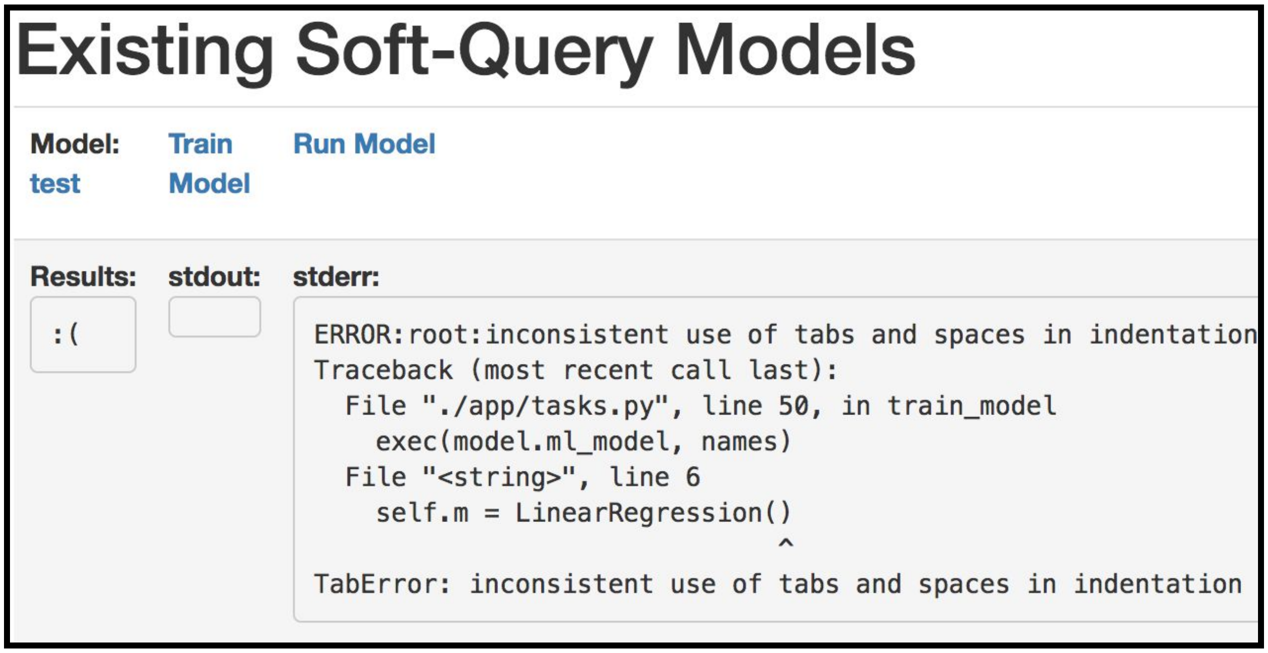
Navigation

A top bar. Both "Soft-Query MVP" and "Home" brings us the the front page. "Create New Model" brings us to a new model page, which is similar to a "Modify model" page.



Front page

The front page contains the list of modules. Each model we can train, predict and modify. We also see the result of the last train/predict run:



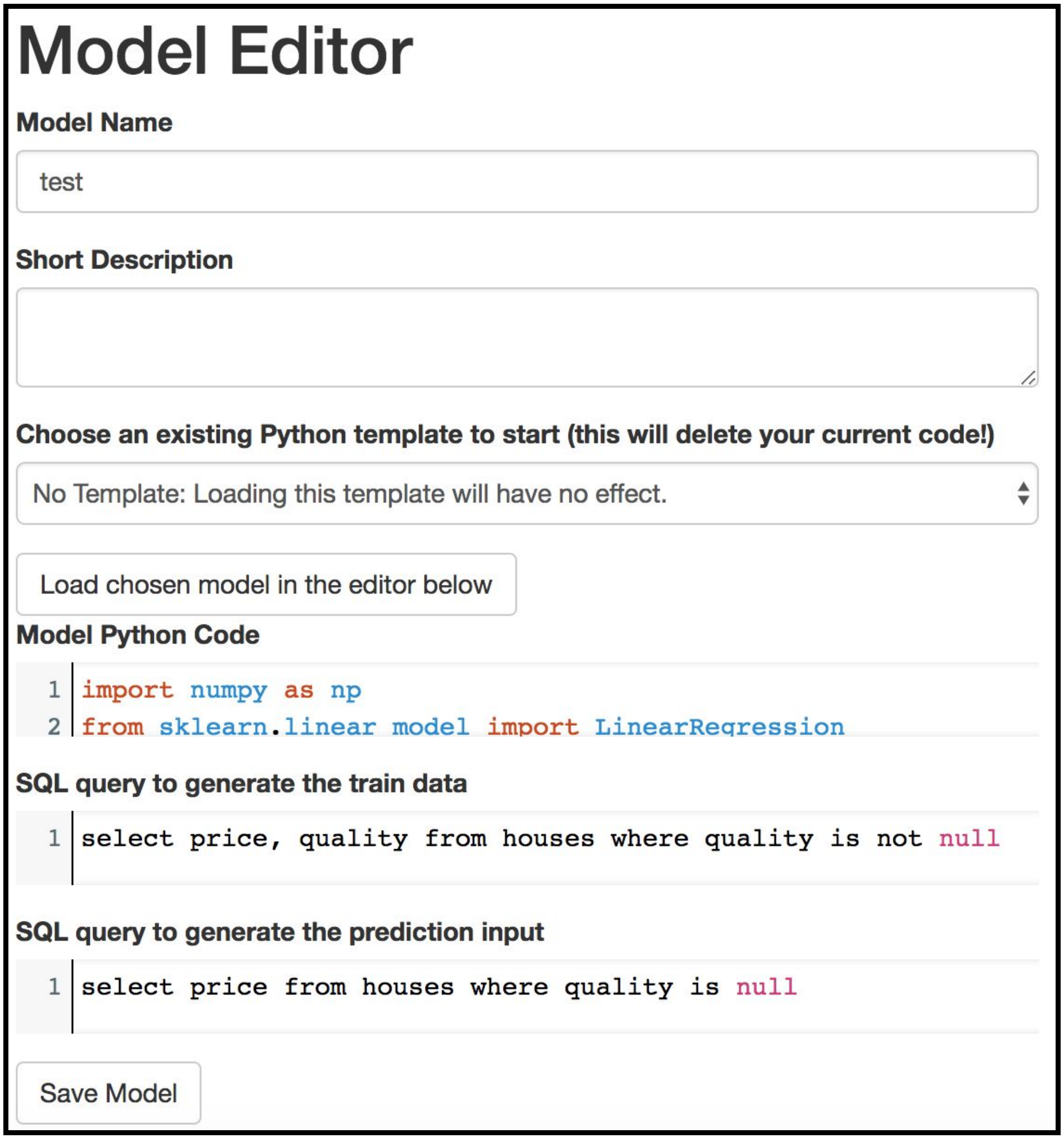
```
ERROR:root:inconsistent use of tabs and spaces in indentation
Traceback (most recent call last):
  File "./app/tasks.py", line 50, in train_model
    exec(model.ml_model, names)
  File "<string>", line 6
    self.m = LinearRegression()
                              ^
TabError: inconsistent use of tabs and spaces in indentation
```

TODO

- "Stop" button
- "Delete Model" button (perhaps better through the model edit page)
- "Duplicate Model" button (perhaps better through the model edit page)
- Style the page nicely
- Collapse long outputs so it doesn't clutter the page.

Model editor

The model editor allows modifying a model:
- Name
- Description (multiline)
- Model code (python), with an option to load an existing model
- SQL query for train
- SQL query for prediction
- Save button



Model Editor

Model Name

test

Short Description

Choose an existing Python template to start (this will delete your current code!)

No Template: Loading this template will have no effect.

Model Python Code

```python
import numpy as np
from sklearn.linear_model import LinearRegression
```

SQL query to generate the train data

```sql
select price, quality from houses where quality is not null
```

SQL query to generate the prediction input

```sql
select price from houses where quality is null
```

TODO

- Automatic syntax check of the python code
- Automatic check of provided model (signature of train/predict functions)
- inputs should be auto-resizeable
- Add model templates
- Add delete/duplicate button

Model templates

Backend

Web server

python3 with flask, based on microblog. For code textboxes we use CodeMirror.

Models

Saved in a sqlite DB. They contain all the text fields from the webpage, along with the trained model state (pickle) and the last result of a train / predict run.

Jobs / Tasks

A task is a background task that we give a worker to do. The workers are implemented using Redis.

train_model

Trains a specific model. The job will execute a sql query, train the model and save (pickle) the trained model in a DB.

run_model

Runs a specific model over some input. The job will execute a sql query, load the model, run it on the sql data, and finally save it to another sql table: `<classifier_name>_prediction`. We may want to let the user modify the table name.

Saving model state

We need a method of saving/loading the model state. We use python's pickle.dumps() and save it to the models DB. A small catch is that class attributes are **NOT** pickled. Read more [here].
The following, for example, will not work:

```
class Model:
  m = LinearRegression()
  def train(self, X, y):
    self.m.fit(X, y)
  def predict(self, X):
    return self.m.predict(X)
```

This is okay, though:

```
class Model:
  def __main__(self):
    self.m = LinearRegression()
  def train(self, X, y):
    self.m.fit(X, y)
  def predict(self, X):
    return self.m.predict(X)
```

Another option is sklearn's dump(), but it only works on sklearn models.

As the model class is dynamically generated, it can not be easily pickled. Using a small hack, we can pickle and unpickle it:

```
names = {}
exec(model_str, names)
global Model
Model = names['Model']
Model.__module__ = __name__
obj = Model()
obj.train(X, y)
state = pickle.dumps(obj)
```

And to unpickle:

```
names = {}
exec(model.ml_model, names)
global Model
Model = names['Model']
Model.__module__ = __name__
obj = pickle.loads(model.state_pkl)
y = obj.predict(X)
```

Predefined Models

Linear Regression

```
from sklearn.linear_model import LinearRegression

class Model:
  def __init__(self):
    self.m = LinearRegression()
  def train(self, X, y):
    self.m.fit(X, y)
  def predict(self, X):
    return self.m.predict(X)
```

Usage

1. Get code and install dependencies from requirements.txt
2. Start redis server:
   `$ redis-server`
3. Start a worker (within environment):
   `$ rq worker`
4. Start webserver:
   `$ FLASK_APP=soft_query_app.py DEBUG_MODE=1 flask run`
5. Profit

Other

Redis

Installing redis on OSX

1. Install redis:
   `$ brew install redis`
   `$ brew services start redis`
2. Install python redis:
   `$ pip3 install rq`
3. Start redis server:
   `$ redis-server`
4. Start a worker (within environment):
   `$ rq worker`
5. Submit job through python:

```
from redis import Redis
from rq import Queue
import time

q = Queue(connection=Redis())

from rq_test_func import delayed_return
job = q.enqueue(delayed_return, 1, 'http://nvie.com', timeout=1)
print(job.result)
time.sleep(2)
print(job.result)
```

6. Run the above code:
   `$ python ./test.py`

7. View worker status:
   ```
   $ rq info
   ```

Possible use case

```
-- Training Input
select user.id, user.embedding, user.cocacola_label -- we need the name of the
user.id column, to create the new table later but not really all the IDs.
where cocacola_label <> ''
from users

-- Inference Input
-- Export stuff to table "<classifier_name>_prediction" with a schema sort of like
(user_id, prediction_result)

select user.id, user.embedding
where cocacola_label == ''
```