Deploy Machine Learning Models with Django

Version 1.0 (04/11/2019)

Piotr Płoński

Introduction

The demand for Machine Learning (ML) applications is growing. Many resources show how to train ML algorithms. However, the ML algorithms work in two phases:

- *the training phase* - in which the ML algorithm is trained based on historical data,
- *the inference phase* - the ML algorithm is used for computing predictions on new data with unknown outcomes.

The benefits for business are in the interference phase when ML algorithms provide information before it is known. There is a technological challenge on how to provide ML algorithms for inference into production systems. There are many requirements which need to be fulfilled:

- ML algorithms deployment automation,
- continuous-integration,
- reproducibility of algorithms and predictions,
- diagnostic and monitoring of algorithms in production,
- governance and regulatory compliance,
- scalability,
- users collaboration.

There are many ways of how ML algorithms can be used:

- The simplest approach is to run the ML algorithm locally to compute predictions on prepared test data and share predictions with others. This approach is easy and fast in implementation. However, it has many drawbacks. It is hard to govern, monitor, scale and collaborate.
- The second, similar approach, is to hard-code the ML algorithm in the system's code. This solution is rather for simple ML algorithms, like Decision Trees or Linear Regression (which are easy to implement independently of the programming language). This solution behaves similar to the first approach - it is easy to implement with many drawbacks.
- The third solution, it to make the ML algorithm available by REST API, RPC or WebSockets. This method requires the implementation of the server which handles requests and forwards them to ML algorithms. In this approach, all requirements for the ML production system can be fulfilled.
- The last solution is to use a commercial vendor for deploying ML algorithms - it can be in the cloud or on-premise. Sometimes, this can be a good solution. When you have a standard ML algorithm so the vendor can handle it and you have money to pay to the vendor (it can be pricy).

This tutorial provides code examples on how to build your ML system available with REST API. In this book, for building the ML service I will use Python 3.6 and Django 2.2.4. This book is the first part that covers the basics which should be enough to build your ML system which:

- can handle many API endpoints,
- each API endpoint can have several ML algorithms with different versions,
- ML code and artifacts (files with ML parameters) are stored in the code repository (git),
- supports fast deployments and continuous integration (tests for both: server and ML code),

- supports monitoring and algorithm diagnostic (support A/B tests),
- is scalable (deployed with containers),
- has a user interface.

There are many ways in which this tutorial can be extended, for example:

- running long jobs for batch predictions or algorithm training with Celery,
- running scheduled jobs with Celery,
- WebSocket interface for Internet-of-Things applications (with Django Channels),
- authentication and user management.

Right now, the above topics are not covered in this tutorial. I will consider writing them in the future based on the reader's feedback. You can send me feedback using this form (https://docs.google.com/forms/d/e/1FAIpQLSfdc8xWZKnIBZojw69rPR2ItKPwU-14-n_-ZNNh-A_kHDmg2A/viewform?usp=sf_link).

In my opinion, building your ML system has a great advantage - it is tailored to your needs. It has all features that are needed in your ML system and can be as complex as you wish.

This tutorial is for readers who are familiar with ML and would like to learn how to build ML web services. Basic Python knowledge is required. The full code of this tutorial is available at: https://github.com/pplonski/my_ml_service (https://github.com/pplonski/my_ml_service).

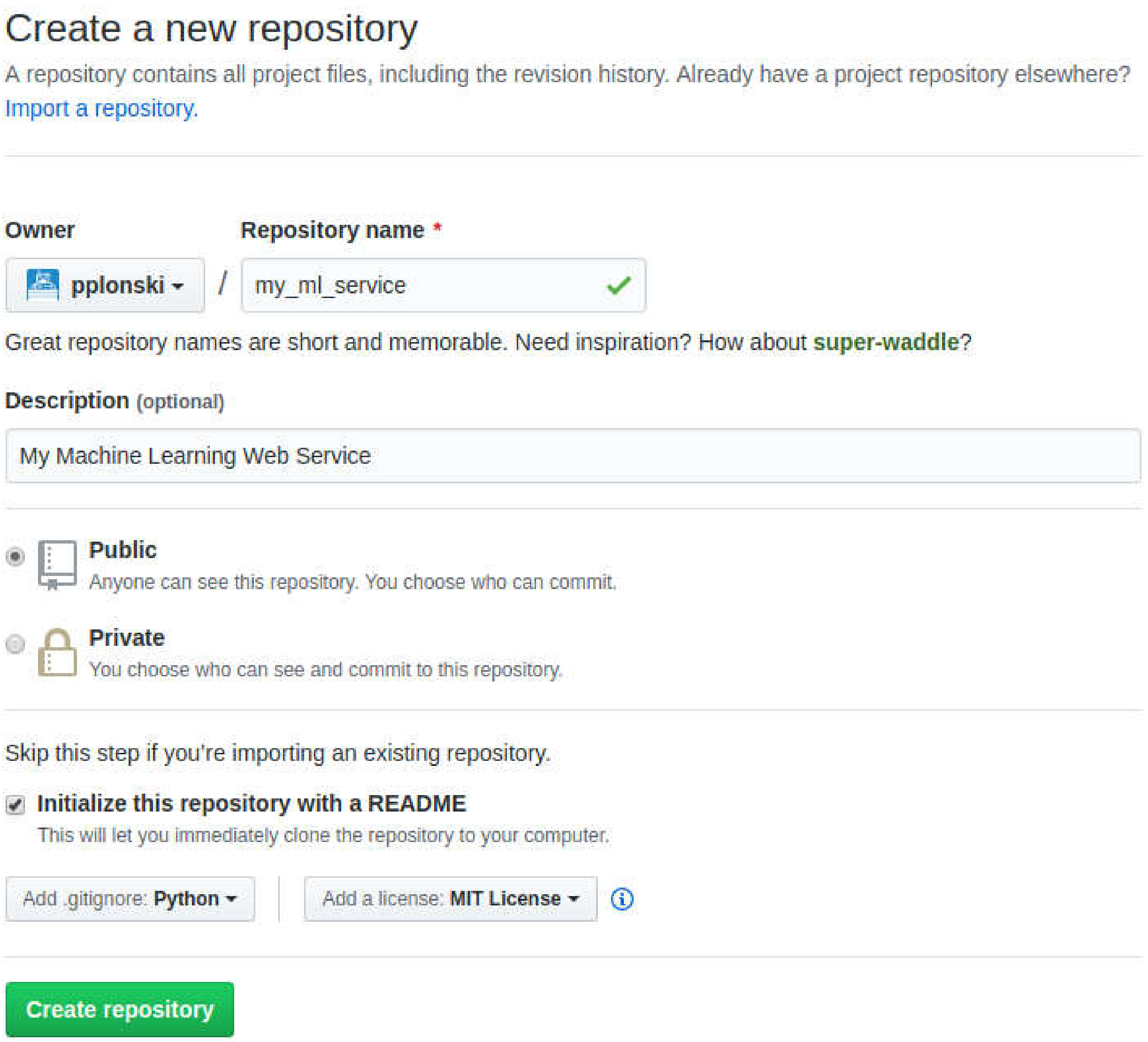
Start

What you will learn in this chapter:

- how to set up a git repository,
- setup environment for development (I will use Ubuntu 16.04),
- install required packages,
- start the Django project.

Setup git repository

To set up a git repository I use GitHub (https://github.com) (it is free for public and private projects). If you have an account there please go to https://github.com/new (https://github.com/new) and set the repository, like in the image (1).



Create a new repository

A repository contains all project files, including the revision history. Already have a project repository elsewhere? Import a repository.

Owner: pplonski / Repository name *: my_ml_service ✓

Great repository names are short and memorable. Need inspiration? How about super-waddle?

Description (optional): My Machine Learning Web Service

● Public
Anyone can see this repository. You choose who can commit.

○ Private
You choose who can see and commit to this repository.

Skip this step if you're importing an existing repository.

☑ Initialize this repository with a README
This will let you immediately clone the repository to your computer.

Add .gitignore: Python | Add a license: MIT License

Figure 1: Setup a new project in github

The full code of this tutorial is available at: https://github.com/pplonski/my_ml_service (https://github.com/pplonski/my_ml_service).

Then please go to your terminal and set the repository:

```
git clone https://github.com/pplonski/my_ml_service.git
cd my_ml_service
ls -l
```

In my case, I had two files in the repository, `LICENSE` and `README.md`.

Installation

Let's set up and activate the environment for development (I'm using Ubuntu 16.04). I will use `virtualenv`:

```
virtualenv venv --python=python3.6
source venv/bin/activate
```

You will need to activate the environment every time you are starting work on your project in the new terminal.

To install needed packages I will use `pip3`:

```
pip3 install django==2.2.4
```

The Django is installed in version `2.2.4`.

Start Django project

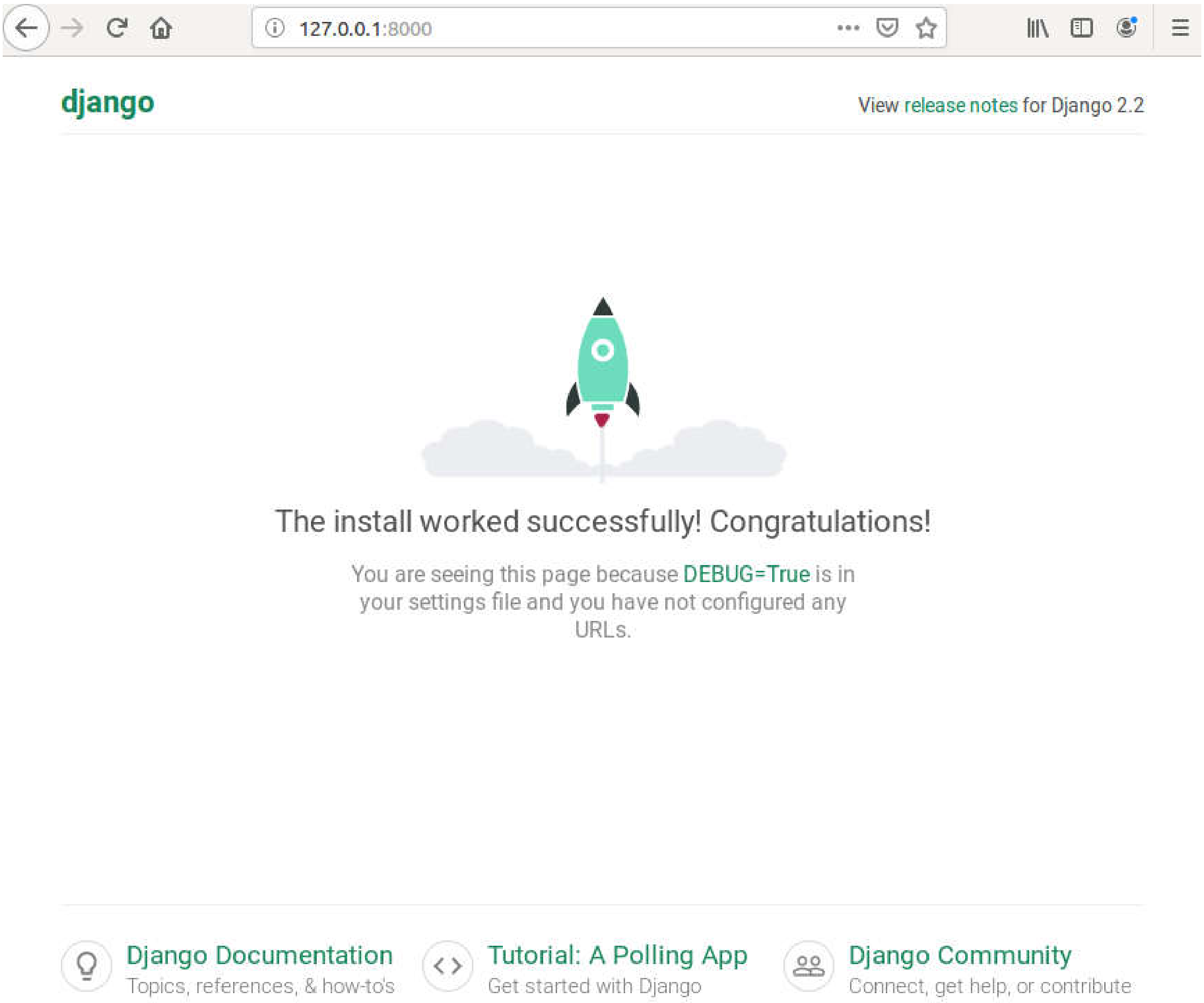
I will set up the Django project in the `backend` directory. The Django project name is set to `server`.

```
mkdir backend
cd backend
django-admin startproject server
```

You can run your initiated server with the following command:

```
cd server
python manage.py runserver
```

When you enter `127.0.0.1:8000` in your favorite web browser you should see default Django welcome site (2).



The install worked successfully! Congratulations!

You are seeing this page because DEBUG=True is in your settings file and you have not configured any URLs.

Figure 2: Django default welcome site

Congratulations!!! you have successfully set up the environment.

Add source files to the repository

Before we go to the next chapter, let's commit new files.

```
# please execute it in your main project directory
git add backend/
git commit -am "setup django project"
git push
```

The following files should be added to your project:

```
new file:   backend/server/manage.py
new file:   backend/server/server/__init__.py
new file:   backend/server/server/settings.py
new file:   backend/server/server/urls.py
new file:   backend/server/server/wsgi.py
```

In your directory, there are other files which are not added to the repository because there are excluded in `.gitignore` file.

Build ML algorithms

In this chapter you will learn:

- how to setup Jupyter notebook,
- how to build two ML algorithms,
- save pre-processing details and algorithms.

Setup Jupyter notebook

For building ML algorithms I'm using Jupyter notebook. It can be easily installed:

```
# run commands in your project directory
pip3 install jupyter notebook
```

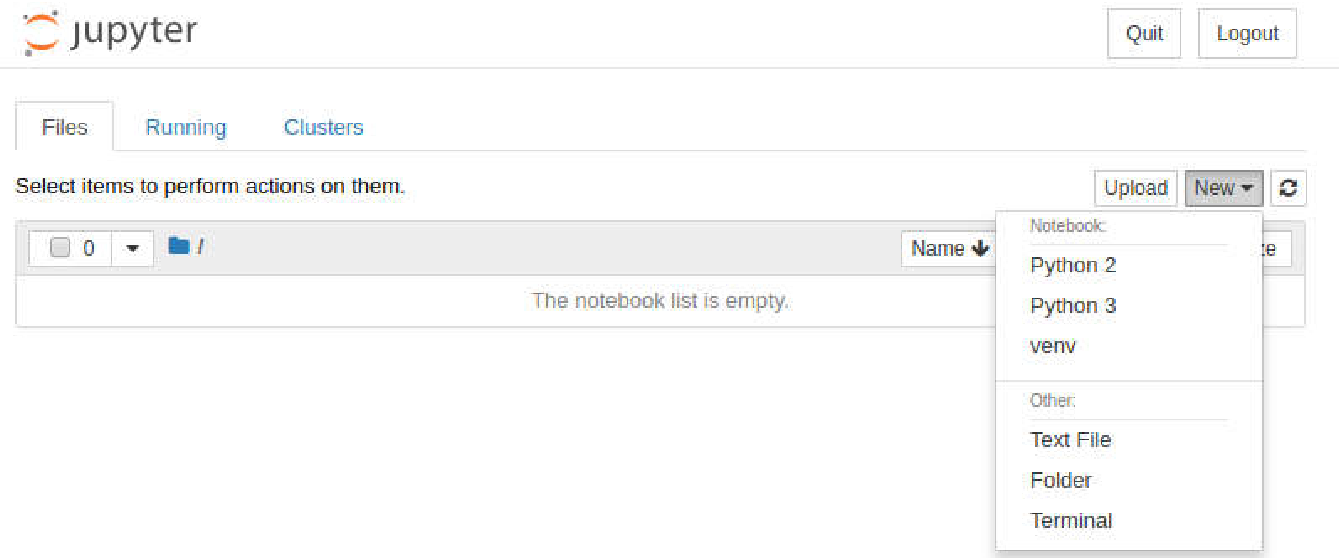
To set Jupyter to use local `virtualenv` environment run:

```
ipython kernel install --user --name=venv
```

I will create a `research` directory where I will put Jupiter files. To start Jupyter notebook run:

```
# create a research directory
mkdir research
cd research
# start Jupyter
jupyter notebook
```

When starting a new notebook make sure that you select the correct kernel, `venv` in our case (image 3).



Select items to perform actions on them.

The notebook list is empty.

Figure 3: Start new jupyter notebook

Train ML algorithms

Before building ML algorithms we need to install packages:

```
pip3 install numpy pandas sklearn joblib
```

The `numpy` and `pandas` packages are used for data manipulation. The `joblib` is used for ML objects saving. Whereas, the `sklearn` package offers a wide range of ML algorithms. We need to reload Jupyter after installation.

The first step in our code is to load packages:

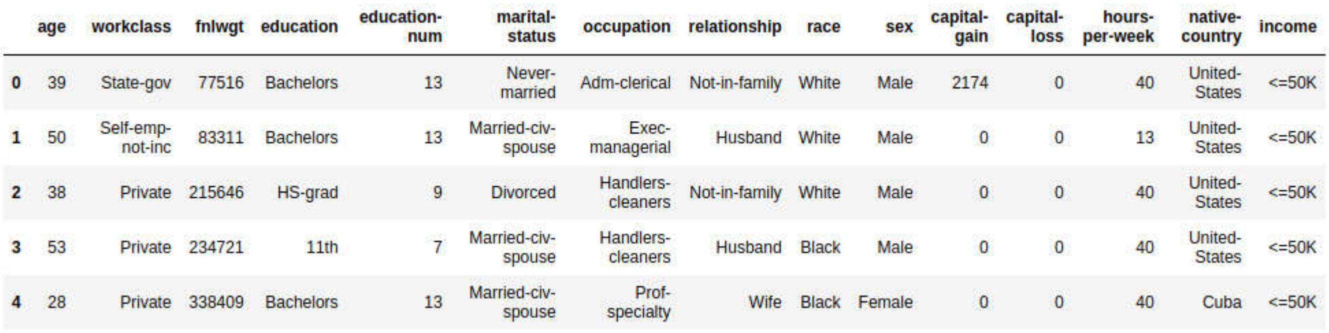
```
import json # will be needed for saving preprocessing details
import numpy as np # for data manipulation
import pandas as pd # for data manipulation
from sklearn.model_selection import train_test_split # will be used for data split
from sklearn.preprocessing import LabelEncoder # for preprocessing
from sklearn.ensemble import RandomForestClassifier # for training the algorithm
from sklearn.ensemble import ExtraTreesClassifier # for training the algorithm
import joblib # for saving algorithm and preprocessing objects
```

Loading data

In this tutorial, I will use Adult Income data set (https://archive.ics.uci.edu/ml/datasets/adult). In this data set, the ML will be used to predict whether income exceeds $50K/year based on census data. I will load data from my public repository with data sets good for start with ML (https://github.com/pplonski/datasets-for-start).

Code to load data and show first rows of data (figure 4):

```
# load dataset
df = pd.read_csv('https://raw.githubusercontent.com/pplonski/datasets-for-start/mas
x_cols = [c for c in df.columns if c != 'income']
# set input matrix and target column
X = df[x_cols]
y = df['income']
# show first rows of data
df.head()
```



	age	workclass	fnlwgt	education	education-num	marital-status	occupation	relationship	race	sex	capital-gain	capital-loss	hours-per-week	native-country	income
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174	0	40	United-States	<=50K
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0	0	13	United-States	<=50K
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0	0	40	United-States	<=50K
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0	0	40	United-States	<=50K
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0	0	40	Cuba	<=50K

Figure 4: First rows of our dataset

The X matrix has 32,561 rows and 14 columns. This is input data for our algorithm, each row describes one person. The y vector has 32,561 values indicating whether income exceeds 50K per year.

Before starting data preprocessing we will split our data into training, and testing subsets. We will use 30% of the data for testing.

```
# data split train / test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_s
```

Data pre-processing

In our data set, there are missing values and categorical columns. For ML algorithm training I will use the `Random Forest` algorithm from the `sklearn` package. In the current implementation it can not handle missing values and categorical columns, that's why we need to apply pre-processing algorithms.

To fill missing values we will use the most frequent value in each column (there are many other filling methods, the one I select is just for example purposes).

```
# fill missing values
train_mode = dict(X_train.mode().iloc[0])
X_train = X_train.fillna(train_mode)
print(train_mode)
```

The `train_mode` values look like:

```
{'age': 31.0,
 'workclass': 3.0,
 'fnlwgt': 121124.0,
 'education': 11.0,
 'education-num': 9.0,
 'marital-status': 2.0,
 'occupation': 9.0,
 'relationship': 0.0,
 'race': 4.0,
 'sex': 1.0,
 'capital-gain': 0.0,
 'capital-loss': 0.0,
 'hours-per-week': 40.0,
 'native-country': 37.0}
```

From `train_mode` you see, that for example in the `age` column the most frequent value is `31.0`.

Let's convert categoricals into numbers. I will use `LabelEncoder` from `sklearn` package:

```
# convert categoricals
encoders = {}
for column in ['workclass', 'education', 'marital-status',
                'occupation', 'relationship', 'race',
                'sex','native-country']:
    categorical_convert = LabelEncoder()
    X_train[column] = categorical_convert.fit_transform(X_train[column])
    encoders[column] = categorical_convert
```

Algorithms training

Data is ready, so we can train our Random Forest algorithm.

```
# train the Random Forest algorithm
rf = RandomForestClassifier(n_estimators = 100)
rf = rf.fit(X_train, y_train)
```

We will also train `Extra Trees` algorithm:

```
# train the Extra Trees algorithm
et = ExtraTreesClassifier(n_estimators = 100)
et = et.fit(X_train, y_train)
```

As you see, training the algorithm is easy, just 2 lines of code - much less than data reading and pre-processing. Now, let's save the algorithm that we have created. The important thing to notice is that the ML algorithm is not only the `rf` and `et` variable (with model weights), but we also need to save pre-processing variables `train_mode` and `encoders` as well. For saving, I will use `joblib` package.

```
# save preprocessing objects and RF algorithm
joblib.dump(train_mode, "./train_mode.joblib", compress=True)
joblib.dump(encoders, "./encoders.joblib", compress=True)
joblib.dump(rf, "./random_forest.joblib", compress=True)
joblib.dump(et, "./extra_trees.joblib", compress=True)
```

Add ML code and artifacts to the repository

Before continuing to the next chapter, let's add our notebook and files to the repository.

```
# execute in project main directory
git add research/*
git commit -am "add ML code and algorithms"
git push
```

Each file with preprocessing objects and algorithms is smaller than 100 MB, which is the GitHub file limit. For larger files it will be better to use separate version control systems like DVC (https://dvc.org) - however, this is a more advanced topic.

Django models

What have you already accomplished:

- you have default Django project initialized,
- you have two ML algorithms trained and ready for inference.

What you will learn in this chapter:

- build Django models to store information about ML algorithms and requests in the database,
- write REST API for your ML algorithms with the `Django REST Framework`.

Create Django models

To create Django models we need to create a new app:

```
# run this in backend/server directory
python manage.py startapp endpoints
mkdir apps
mv endpoints/ apps/
```

With the above commands, we have created the `endpoints` app and moved it to the `apps` directory. I have added the `apps` directory to keep the project clean.

```
# list files in apps/endpoints
ls apps/endpoints/
# admin.py  apps.py  __init__.py  migrations  models.py  tests.py  views.py
```

Let's go to `apps/endpoints/models.py` file and define database models (Django provides object-relational mapping layer (ORM)).

```
from django.db import models

class Endpoint(models.Model):
    '''
    The Endpoint object represents ML API endpoint.

    Attributes:
        name: The name of the endpoint, it will be used in API URL,
        owner: The string with owner name,
        created_at: The date when endpoint was created.
    '''
    name = models.CharField(max_length=128)
    owner = models.CharField(max_length=128)
    created_at = models.DateTimeField(auto_now_add=True, blank=True)

class MLAlgorithm(models.Model):
    '''
    The MLAlgorithm represent the ML algorithm object.

    Attributes:
        name: The name of the algorithm.
        description: The short description of how the algorithm works.
        code: The code of the algorithm.
        version: The version of the algorithm similar to software versioning.
        owner: The name of the owner.
        created_at: The date when MLAlgorithm was added.
        parent_endpoint: The reference to the Endpoint.
    '''
    name = models.CharField(max_length=128)
    description = models.CharField(max_length=1000)
    code = models.CharField(max_length=50000)
    version = models.CharField(max_length=128)
    owner = models.CharField(max_length=128)
    created_at = models.DateTimeField(auto_now_add=True, blank=True)
    parent_endpoint = models.ForeignKey(Endpoint, on_delete=models.CASCADE)

class MLAlgorithmStatus(models.Model):
    '''
    The MLAlgorithmStatus represent status of the MLAlgorithm which can change duri

    Attributes:
        status: The status of algorithm in the endpoint. Can be: testing, staging,
        active: The boolean flag which point to currently active status.
        created_by: The name of creator.
        created_at: The date of status creation.
        parent_mlalgorithm: The reference to corresponding MLAlgorithm.

    '''
    status = models.CharField(max_length=128)
    active = models.BooleanField()
    created_by = models.CharField(max_length=128)
    created_at = models.DateTimeField(auto_now_add=True, blank=True)
    parent_mlalgorithm = models.ForeignKey(MLAlgorithm, on_delete=models.CASCADE, r

class MLRequest(models.Model):
    '''
    The MLRequest will keep information about all requests to ML algorithms.

    Attributes:
        input_data: The input data to ML algorithm in JSON format.
        full_response: The response of the ML algorithm.
        response: The response of the ML algorithm in JSON format.
        feedback: The feedback about the response in JSON format.
        created_at: The date when request was created.
        parent_mlalgorithm: The reference to MLAlgorithm used to compute response.
    '''
    input_data = models.CharField(max_length=10000)
    full_response = models.CharField(max_length=10000)
    response = models.CharField(max_length=10000)
    feedback = models.CharField(max_length=10000, blank=True, null=True)
    created_at = models.DateTimeField(auto_now_add=True, blank=True)
    parent_mlalgorithm = models.ForeignKey(MLAlgorithm, on_delete=models.CASCADE)
```

We defined three models:

- `Endpoint` - to keep information about our endpoints,
- `MLAlgorithm` - to keep information about ML algorithms used in the service,
- `MLAlgorithmStatus` - to keep information about ML algorithm statuses. The status can change in time, for example, we can set *testing* as initial status and then after testing period switch to *production* state.
- `MLRequest` - to keep information about all requests to ML algorithms. It will be needed to monitor ML algorithms and run A/B tests.

We need to add our app to `INSTALLED_APPS` in `backend/server/server/settings.py`, it should look like:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    # apps
    'apps.endpoints'
]
```

To apply our models to the database we need to run migrations:

```
# please run it in backend/server directory
python manage.py makemigrations
python manage.py migrate
```

The above commands will create tables in the database. By default, Django is using SQLite as a database. For this tutorial, we can keep this simple database, for more advanced projects you can set a Postgres or MySQL as a database (you can configure this by setting `DATABASES` variable in `backend/server/server/settings.py`).

Create REST API for models

So far we have defined database models, but we will not see anything new when running the web server. We need to specify REST API to our objects. The simplest and cleanest way to achieve this is to use `Django REST Framework` (https://www.django-rest-framework.org/) (DRF). To install `DRF` we need to run:

```
pip3 install djangorestframework
pip3 install markdown       # Markdown support for the browsable API.
pip3 install django-filter  # Filtering support
```

and add it to `INSTALLED_APPS` in `backend/server/server/settings.py`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework', # add django rest framework
    # apps
    'apps.endpoints'
]
```

To see something in the browser we need to define:

- serializers - they will define how database objects are mapped in requests,
- views - how our models are accessed in REST API,
- urls - definition of REST API URL addresses for our models.

DRF Serializers

Please add `serializers.py` file to `server/apps/endpoints` directory:

```
# backend/server/apps/endpoints/serializers.py file
from rest_framework import serializers
from apps.endpoints.models import Endpoint
from apps.endpoints.models import MLAlgorithm
from apps.endpoints.models import MLAlgorithmStatus
from apps.endpoints.models import MLRequest

class EndpointSerializer(serializers.ModelSerializer):
    class Meta:
        model = Endpoint
        read_only_fields = ("id", "name", "owner", "created_at")
        fields = read_only_fields


class MLAlgorithmSerializer(serializers.ModelSerializer):

    current_status = serializers.SerializerMethodField(read_only=True)

    def get_current_status(self, mlalgorithm):
        return MLAlgorithmStatus.objects.filter(parent_mlalgorithm=mlalgorithm).lat

    class Meta:
        model = MLAlgorithm
        read_only_fields = ("id", "name", "description", "code",
                            "version", "owner", "created_at",
                            "parent_endpoint", "current_status")
        fields = read_only_fields

class MLAlgorithmStatusSerializer(serializers.ModelSerializer):
    class Meta:
        model = MLAlgorithmStatus
        read_only_fields = ("id", "active")
        fields = ("id", "active", "status", "created_by", "created_at",
                            "parent_mlalgorithm")

class MLRequestSerializer(serializers.ModelSerializer):
    class Meta:
        model = MLRequest
        read_only_fields = (
            "id",
            "input_data",
            "full_response",
            "response",
            "created_at",
            "parent_mlalgorithm",
        )
        fields =  (
            "id",
            "input_data",
            "full_response",
            "response",
            "feedback",
            "created_at",
            "parent_mlalgorithm",
        )
```

Serializers will help with packing and unpacking database objects into JSON objects. In `Endpoints` and `MLAlgorithm` serializers, we defined all read-only fields. This is because, we will create and modify our objects only on the server-side.For `MLAlgorithmStatus`, fields `status`, `created_by`, `created_at` and `parent_mlalgorithm` are in read and write mode, we will use the to set algorithm status by REST API. For `MLRequest` serializer there is a `feedback` field that is left in read and write mode - it will be needed to provide feedback about predictions to the server.

The `MLAlgorithmSerializer` is more complex than others. It has one filed `current_status` that represents the latest status from `MLAlgorithmStatus`.

Views

To add views please open `backend/server/endpoints/views.py` file and add the following code:

```
# backend/server/apps/endpoints/views.py file
from rest_framework import viewsets
from rest_framework import mixins

from apps.endpoints.models import Endpoint
from apps.endpoints.serializers import EndpointSerializer

from apps.endpoints.models import MLAlgorithm
from apps.endpoints.serializers import MLAlgorithmSerializer

from apps.endpoints.models import MLAlgorithmStatus
from apps.endpoints.serializers import MLAlgorithmStatusSerializer

from apps.endpoints.models import MLRequest
from apps.endpoints.serializers import MLRequestSerializer

class EndpointViewSet(
    mixins.RetrieveModelMixin, mixins.ListModelMixin, viewsets.GenericViewSet
):
    serializer_class = EndpointSerializer
    queryset = Endpoint.objects.all()


class MLAlgorithmViewSet(
    mixins.RetrieveModelMixin, mixins.ListModelMixin, viewsets.GenericViewSet
):
    serializer_class = MLAlgorithmSerializer
    queryset = MLAlgorithm.objects.all()


def deactivate_other_statuses(instance):
    old_statuses = MLAlgorithmStatus.objects.filter(parent_mlalgorithm = instance.p
                                                        created_at__lt=instance.cre
                                                        active=True)
    for i in range(len(old_statuses)):
        old_statuses[i].active = False
    MLAlgorithmStatus.objects.bulk_update(old_statuses, ["active"])

class MLAlgorithmStatusViewSet(
    mixins.RetrieveModelMixin, mixins.ListModelMixin, viewsets.GenericViewSet,
    mixins.CreateModelMixin
):
    serializer_class = MLAlgorithmStatusSerializer
    queryset = MLAlgorithmStatus.objects.all()
    def perform_create(self, serializer):
        try:
            with transaction.atomic():
                instance = serializer.save(active=True)
                # set active=False for other statuses
                deactivate_other_statuses(instance)



        except Exception as e:
            raise APIException(str(e))

class MLRequestViewSet(
    mixins.RetrieveModelMixin, mixins.ListModelMixin, viewsets.GenericViewSet,
    mixins.UpdateModelMixin
):
    serializer_class = MLRequestSerializer
    queryset = MLRequest.objects.all()
```

For each model, we created a view which will allow to retrieve single object or list of objects. We will not allow to create or modify `Endpoints`, `MLAlgorithms` by REST API. The code to to handle creation of new ML related objects will be on server side, I will describe it in the next chapter.

We will allow to create `MLAlgorithmStatus` objects by REST API. We don't allow to edit statuses for ML algorithms as we want to keep all status history.

We allow to edit `MLRequest` objects, however only `feedback` field (please take a look at serializer definition).

URLs

The last step is to add URLs to access out models. Please add `urls.py` file in `backend/server/apps/endpoints` with following code:

```
# backend/server/apps/endpoints/urls.py file
from django.conf.urls import url, include
from rest_framework.routers import DefaultRouter

from apps.endpoints.views import EndpointViewSet
from apps.endpoints.views import MLAlgorithmViewSet
from apps.endpoints.views import MLAlgorithmStatusViewSet
from apps.endpoints.views import MLRequestViewSet

router = DefaultRouter(trailing_slash=False)
router.register(r"endpoints", EndpointViewSet, basename="endpoints")
router.register(r"mlalgorithms", MLAlgorithmViewSet, basename="mlalgorithms")
router.register(r"mlalgorithmstatuses", MLAlgorithmStatusViewSet, basename="mlalgor
router.register(r"mlrequests", MLRequestViewSet, basename="mlrequests")

urlpatterns = [
    url(r"^api/v1/", include(router.urls)),
]
```

The above code will create REST API routers to our database models. Our models will be accessed by following the URL pattern:

```
http://<server-ip>/api/v1/<object-name>
```

You can notice that we include `v1` in the API address. This might be needed later for API versioning.

We need to add endpoints urls to main `urls.py` file of the server (file `backend/server/server/urls.py`):

```
# backend/server/server/urls.py file
from django.conf.urls import url, include
from django.contrib import admin
from django.urls import path

from apps.endpoints.urls import urlpatterns as endpoints_urlpatterns

urlpatterns = [
    path('admin/', admin.site.urls),
]

urlpatterns += endpoints_urlpatterns
```

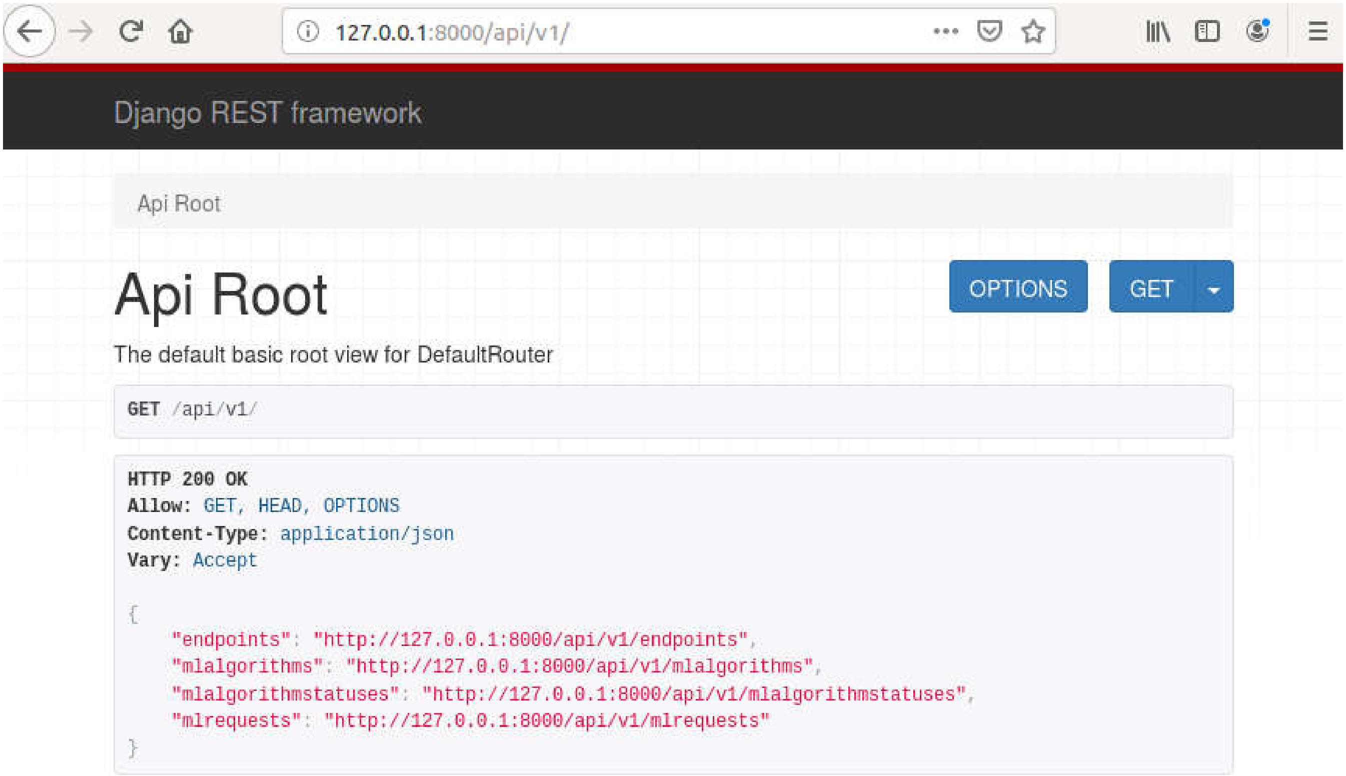
Run the server

We have added many new things, let's check if all works.

Please run the server:

```
# in backend/server
python manage.py runserver
```

and open `http://127.0.0.1:8000/api/v1/` in the web browser. You should see DRF view (image 5).



Api Root

The default basic root view for DefaultRouter

```
GET /api/v1/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "endpoints": "http://127.0.0.1:8000/api/v1/endpoints",
    "mlalgorithms": "http://127.0.0.1:8000/api/v1/mlalgorithms",
    "mlalgorithmstatuses": "http://127.0.0.1:8000/api/v1/mlalgorithmstatuses",
    "mlrequests": "http://127.0.0.1:8000/api/v1/mlrequests"
}
```

Figure 5: Default Django REST Framework view

The DRF provides nice interface, so you can click on any URL and check the objects (for example on http://127.0.0.1:8000/api/v1/endpoints). You should see empty list for all objects, because we didn't add anything there yet. We will add ML algorithms and endpoints in the next chapter.

Add code to the repository

The last step in this chapter is to add a new code to the repository.

```
# please run in backend/server directory
git add apps/endpoints
git commit -am "endpoints models"
git push
```

Add ML algorithms to the server code

So far you have accomplished:

- train two ML algorithms,
- create Django server with database models and REST API endpoints which will represent ML endpoints, models and requests.

What you will learn in this chapter:

- create ML code in the server,
- write ML algorithms registry,
- add ML algorithms to the server.

ML code in the server

In the chapter 3 we have created two ML algorithms (with Random Forest and Extra Trees). They were implemented in the Jupyter notebook. Now, we will write code on the server-side that will use previously trained algorithms. In this chapter we will include on server-side only the Random Forest algorithm (for simplicity).

In the directory `backend/server/apps` let's create new directory `ml` to keep all ML related code and `income_classifier` directory to keep our income classifiers.

```
# please run in backend/server/apps
mkdir ml
mkdir ml/income_classifier
```

In `income_classifier` directory let's add new file `random_forest.py` and empty file `__init__.py`.

In `random_forest.py` file we will implement the ML algorithm code.

```
# file backend/server/apps/ml/income_classifier/random_forest.py
import joblib
import pandas as pd

class RandomForestClassifier:
    def __init__(self):
        path_to_artifacts = "../../research/"
        self.values_fill_missing =  joblib.load(path_to_artifacts + "train_mode.job
        self.encoders = joblib.load(path_to_artifacts + "encoders.joblib")
        self.model = joblib.load(path_to_artifacts + "random_forest.joblib")

    def preprocessing(self, input_data):
        # JSON to pandas DataFrame
        input_data = pd.DataFrame(input_data, index=[0])
        # fill missing values
        input_data.fillna(self.values_fill_missing)
        # convert categoricals
        for column in [
            "workclass",
            "education",
            "marital-status",
            "occupation",
            "relationship",
            "race",
            "sex",
            "native-country",
        ]:
            categorical_convert = self.encoders[column]
            input_data[column] = categorical_convert.transform(input_data[column])

        return input_data

    def predict(self, input_data):
        return self.model.predict_proba(input_data)

    def postprocessing(self, input_data):
        label = "<=50K"
        if input_data[1] > 0.5:
            label = ">50K"
        return {"probability": input_data[1], "label": label, "status": "OK"}

    def compute_prediction(self, input_data):
        try:
            input_data = self.preprocessing(input_data)
            prediction = self.predict(input_data)[0]  # only one sample
            prediction = self.postprocessing(prediction)
        except Exception as e:
            return {"status": "Error", "message": str(e)}

        return prediction
```

The `RandomForestClassifier` algorithm has five methods:

- `__init__` - the constructor which loads preprocessing objects and Random Forest object (created with Jupyter notebook)
- `preprocessing` - the method which takes as input JSON data, converts it to Pandas DataFrame and apply pre-processing
- `predict` - the method that calls ML for computing predictions on prepared data,
- `postprocessing` - the method that applies post-processing on prediction values,
- `compute_prediction` - the method that combines: `preprocessing`, `predict` and `postprocessing` and returns JSON object with the response.

To enable our code in the Django we need to add `ml` app to `INSTALLED_APPS` in `backend/server/server/settings.py`:

```
INSTALLED_APPS = [
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
    'rest_framework',
    # apps
    'apps.endpoints',
    'apps.ml'
]
```

ML code tests

Let's write a test case that will check if our Random Forest algorithm is working as expected. For testing, I will use one row from train data and check if the prediction is correct.

Please add two files into `ml` directory: empty `__init__.py` file and `tests.py` file with the following code:

```
from django.test import TestCase

from apps.ml.income_classifier.random_forest import RandomForestClassifier

class MLTests(TestCase):
    def test_rf_algorithm(self):
        input_data = {
            "age": 37,
            "workclass": "Private",
            "fnlwgt": 34146,
            "education": "HS-grad",
            "education-num": 9,
            "marital-status": "Married-civ-spouse",
            "occupation": "Craft-repair",
            "relationship": "Husband",
            "race": "White",
            "sex": "Male",
            "capital-gain": 0,
            "capital-loss": 0,
            "hours-per-week": 68,
            "native-country": "United-States"
        }
        my_alg = RandomForestClassifier()
        response = my_alg.compute_prediction(input_data)
        self.assertEqual('OK', response['status'])
        self.assertTrue('label' in response)
        self.assertEqual('<=50K', response['label'])
```

The above test is:

- constructing an input JSON data object,
- initializing ML algorithm,
- computing ML prediction and checking the prediction outcome.

To run Django tests run the following command:

```
# please run in backend/server directory
python manage.py test apps.ml.tests
```

You should see that 1 test was run.

```
System check identified no issues (0 silenced).
.
----------------------------------------------------------------------
Ran 1 test in 0.661s

OK
```

Algorithms registry

We have the ML code ready and tested. We need to connect it with the server code. For this, I will create the ML registry object, that will keep information about available algorithms and corresponding endpoints.

Let's add `registry.py` file in the `backend/server/apps/ml/` directory.

```
# file backend/server/apps/ml/registry.py
from apps.endpoints.models import Endpoint
from apps.endpoints.models import MLAlgorithm
from apps.endpoints.models import MLAlgorithmStatus

class MLRegistry:
    def __init__(self):
        self.endpoints = {}

    def add_algorithm(self, endpoint_name, algorithm_object, algorithm_name,
                    algorithm_status, algorithm_version, owner,
                    algorithm_description, algorithm_code):
        # get endpoint
        endpoint, _ = Endpoint.objects.get_or_create(name=endpoint_name, owner=owne

        # get algorithm
        database_object, algorithm_created = MLAlgorithm.objects.get_or_create(
                name=algorithm_name,
                description=algorithm_description,
                code=algorithm_code,
                version=algorithm_version,
                owner=owner,
                parent_endpoint=endpoint)
        if algorithm_created:
            status = MLAlgorithmStatus(status = algorithm_status,
                                        created_by = owner,
                                        parent_mlalgorithm = database_object,
                                        active = True)
            status.save()

        # add to registry
        self.endpoints[database_object.id] = algorithm_object
```

The registry keeps simple `dict` object with a mapping of algorithm id to algorithm object.

To check if the code is working as expected, we can add test case in the `backend/server/apps/ml/tests.py` file:

```
# add at the beginning of the file:
import inspect
from apps.ml.registry import MLRegistry

# ...
# the rest of the code
# ...

# add below method to MLTests class:
    def test_registry(self):
        registry = MLRegistry()
        self.assertEqual(len(registry.endpoints), 0)
        endpoint_name = "income_classifier"
        algorithm_object = RandomForestClassifier()
        algorithm_name = "random forest"
        algorithm_status = "production"
        algorithm_version = "0.0.1"
        algorithm_owner = "Piotr"
        algorithm_description = "Random Forest with simple pre- and post-processing
        algorithm_code = inspect.getsource(RandomForestClassifier)
        # add to registry
        registry.add_algorithm(endpoint_name, algorithm_object, algorithm_name,
                    algorithm_status, algorithm_version, algorithm_owner,
                    algorithm_description, algorithm_code)
        # there should be one endpoint available
        self.assertEqual(len(registry.endpoints), 1)
```

This simple test adds a ML algorithm to the registry. To run tests:

```
# please run in backend/server
python manage.py test apps.ml.tests
```

Tests output:

```
System check identified no issues (0 silenced).
..
----------------------------------------------------------------------
Ran 2 tests in 0.679s

OK
```

Add ML algorithms to the registry

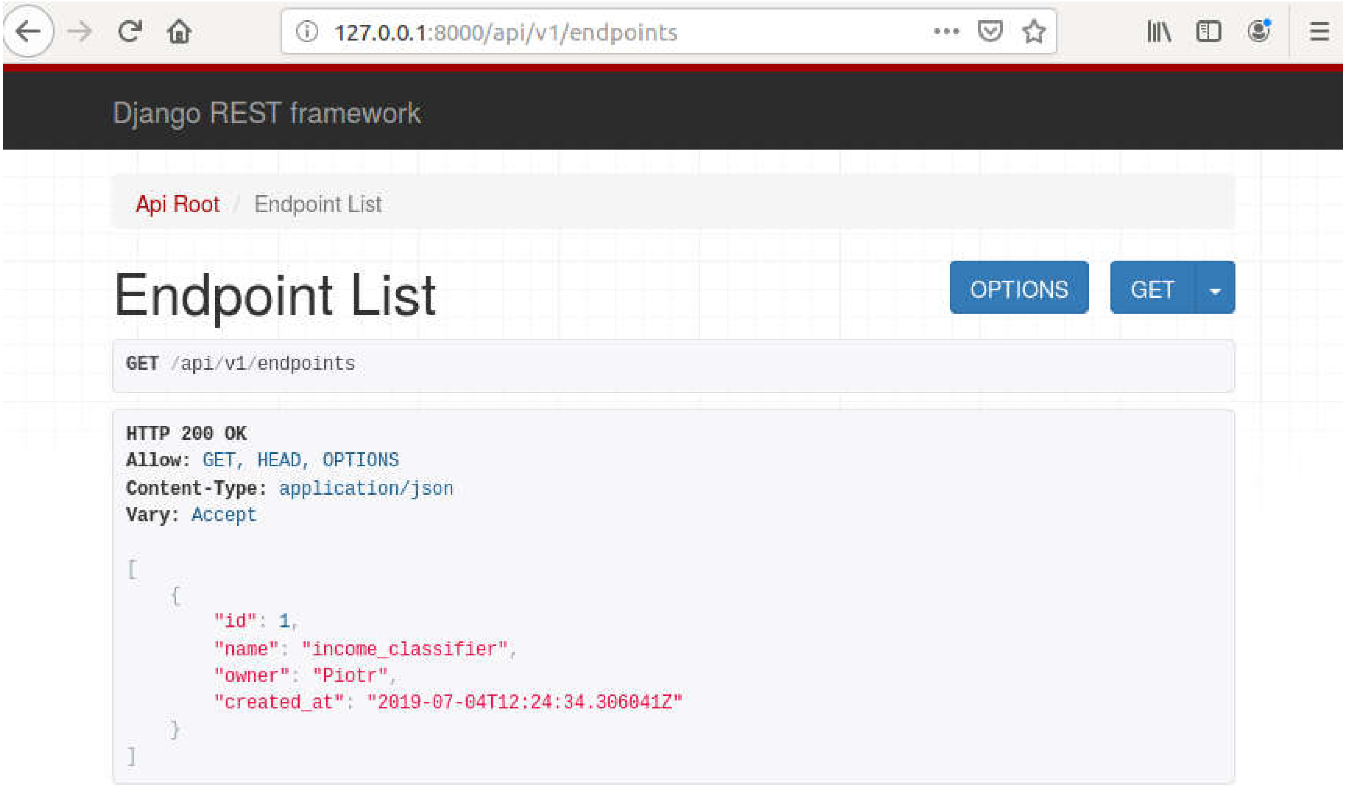
The registry code is ready, we need to specify one place in the server code which will add ML algorithms to the registry when the server is starting. The best place to do it is `backend/server/server/wsgi.py` file. Please set the following code in the file:

```
# file backend/server/server/wsgi.py
import os
from django.core.wsgi import get_wsgi_application
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'server.settings')
application = get_wsgi_application()

# ML registry
import inspect
from apps.ml.registry import MLRegistry
from apps.ml.income_classifier.random_forest import RandomForestClassifier

try:
    registry = MLRegistry() # create ML registry
    # Random Forest classifier
    rf = RandomForestClassifier()
    # add to ML registry
    registry.add_algorithm(endpoint_name="income_classifier",
                            algorithm_object=rf,
                            algorithm_name="random forest",
                            algorithm_status="production",
                            algorithm_version="0.0.1",
                            owner="Piotr",
                            algorithm_description="Random Forest with simple pre- a
                            algorithm_code=inspect.getsource(RandomForestClassifier

except Exception as e:
    print("Exception while loading the algorithms to the registry,", str(e))
```

After starting the server with:

```
python manage.py runserver
```

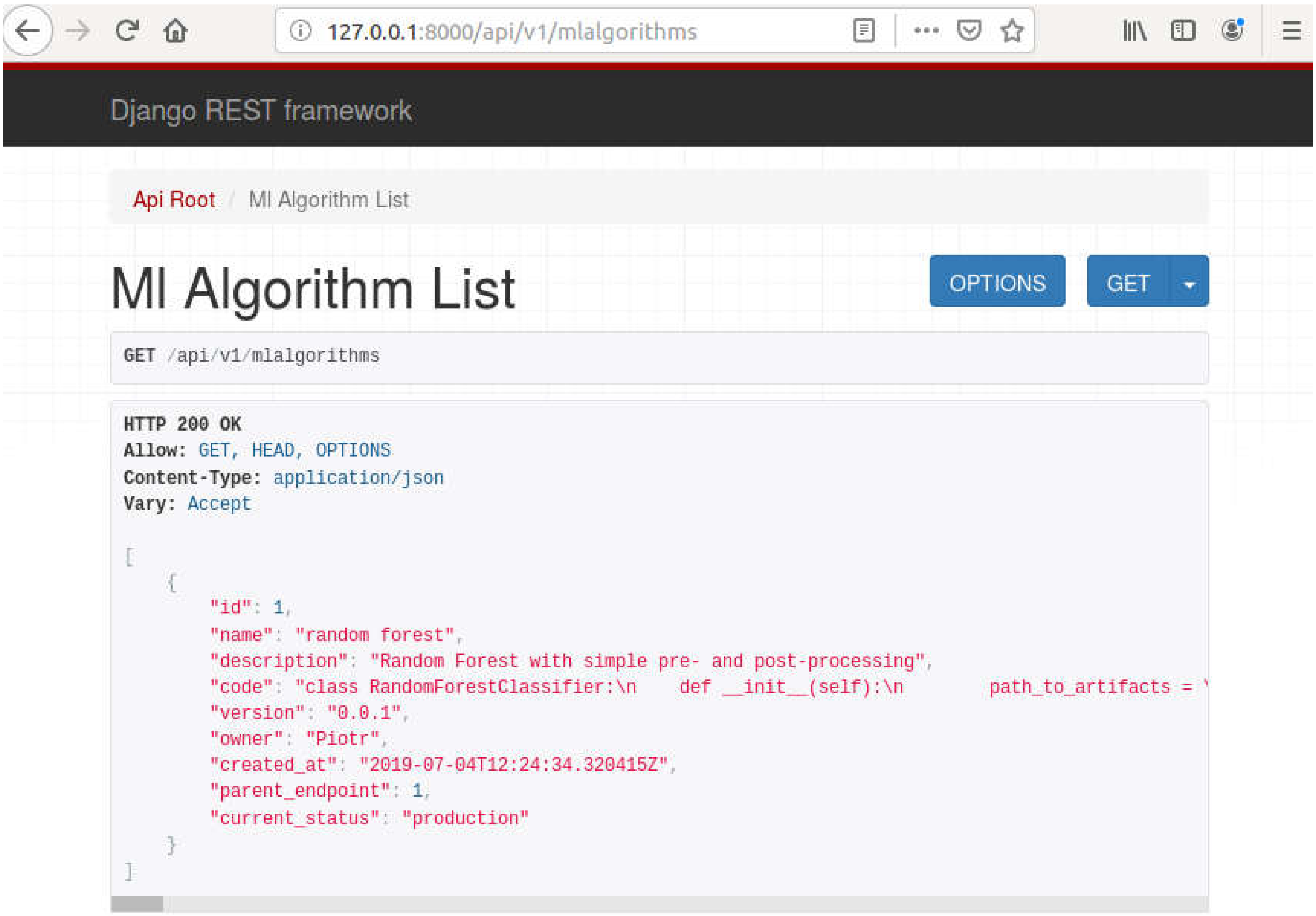
you can check the endpoints and ML algorithms in the browser. At the URL: `http://127.0.0.1:8000/api/v1/endpoints` you can check endpoints (image 6), and at `http://127.0.0.1:8000/api/v1/mlalgorithms` you can check algorithms (image 7).



```
GET /api/v1/endpoints
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "income_classifier",
        "owner": "Piotr",
        "created_at": "2019-07-04T12:24:34.306041Z"
    }
]
```

Figure 6: List of endpoints defined in the service


127.0.0.1:8000/api/v1/mlalgorithms
Django REST framework
Api Root / Ml Algorithm List
Ml Algorithm List
OPTIONS
GET

```
GET /api/v1/mlalgorithms
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "random forest",
        "description": "Random Forest with simple pre- and post-processing",
        "code": "class RandomForestClassifier:\n    def __init__(self):\n        path_to_artifacts = '
        "version": "0.0.1",
        "owner": "Piotr",
        "created_at": "2019-07-04T12:24:34.320415Z",
        "parent_endpoint": 1,
        "current_status": "production"
    }
]
```

Figure 7: List of ML algorithms defined in the service

Add code to repository

We need to commit a new code to the repository.

```
# please run in backend/server directory
git add apps/ml/
git commit -am "add ml code"
git push
```

What's next?

We have our ML algorithm in the database and we can access information about it with REST API, but how to do predictions? This will be the subject of the next chapter.

Making predictions

What have you learned already:

- you have created two ML algorithms in Jupyter notebook,
- you have created Django app with database models and REST API,
- you have added the ML code to the server code and created a ML registry.

What you will learn in this chapter:

- you will add a view for handling requests in the server and forwarding them to ML code,
- you will add API URL to the view,
- you will write tests for predictions.

Predictions view

Firstly, we will create the view for predictions that can accept POST requests with JSON data and forward it to the correct ML algorithm.

In `backend/server/apps/endpoints/views.py` we need to add the following code:

```
# please add imports
import json
from numpy.random import rand
from rest_framework import views, status
from rest_framework.response import Response
from apps.ml.registry import MLRegistry
from server.wsgi import registry

'''
... the rest of the backend/server/apps/endpoints/views.py file ...
'''

class PredictView(views.APIView):
    def post(self, request, endpoint_name, format=None):

        algorithm_status = self.request.query_params.get("status", "production")
        algorithm_version = self.request.query_params.get("version")

        algs = MLAlgorithm.objects.filter(parent_endpoint__name = endpoint_name, st

        if algorithm_version is not None:
            algs = algs.filter(version = algorithm_version)

        if len(algs) == 0:
            return Response(
                {"status": "Error", "message": "ML algorithm is not available"},
                status=status.HTTP_400_BAD_REQUEST,
            )
        if len(algs) != 1 and algorithm_status != "ab_testing":
            return Response(
                {"status": "Error", "message": "ML algorithm selection is ambiguous
                status=status.HTTP_400_BAD_REQUEST,
            )
        alg_index = 0
        if algorithm_status == "ab_testing":
            alg_index = 0 if rand() < 0.5 else 1

        algorithm_object = registry.endpoints[algs[alg_index].id]
        prediction = algorithm_object.compute_prediction(request.data)


        label = prediction["label"] if "label" in prediction else "error"
        ml_request = MLRequest(
            input_data=json.dumps(request.data),
            full_response=prediction,
            response=label,
            feedback="",
            parent_mlalgorithm=algs[alg_index],
        )
        ml_request.save()

        prediction["request_id"] = ml_request.id

        return Response(prediction)
```

Let's add the URL for predictions. The file `backend/server/apps/endpoints/urls.py` should look like below:

```
# file backend/server/apps/endpoints/urls.py

from django.conf.urls import url, include
from rest_framework.routers import DefaultRouter

from apps.endpoints.views import EndpointViewSet
from apps.endpoints.views import MLAlgorithmViewSet
from apps.endpoints.views import MLRequestViewSet
from apps.endpoints.views import PredictView # import PredictView

router = DefaultRouter(trailing_slash=False)
router.register(r"endpoints", EndpointViewSet, basename="endpoints")
router.register(r"mlalgorithms", MLAlgorithmViewSet, basename="mlalgorithms")
router.register(r"mlrequests", MLRequestViewSet, basename="mlrequests")

urlpatterns = [
    url(r"^api/v1/", include(router.urls)),
    # add predict url
    url(
        r"^api/v1/(?P<endpoint_name>.+)/predict$", PredictView.as_view(), name="pre
    ),
]
```

OK, let's go into details. The `PredictView` accepts only POST requests. It is available at:

`https://<server_ip/>api/v1/<endpoint_name>/predict`

The `endpoint_name` is defining the endpoint that we are trying to reach. In our case (in local development) the ML algorithm can be accessed at:

`http://127.0.0.1:8000/api/v1/income_classifier/predict`

The `income_classifier` is the endpoint name (you can check endpoints at `http://127.0.0.1:8000/api/v1/endpoints`).

What is more, you can specify algorithm status or version in the URL. To specify status and version you need to include them in the URL, for example:

`http://127.0.0.1:8000/api/v1/income_classifier/predict?status=testing&version=1.1.1`.

By default, there is a used `production` status.

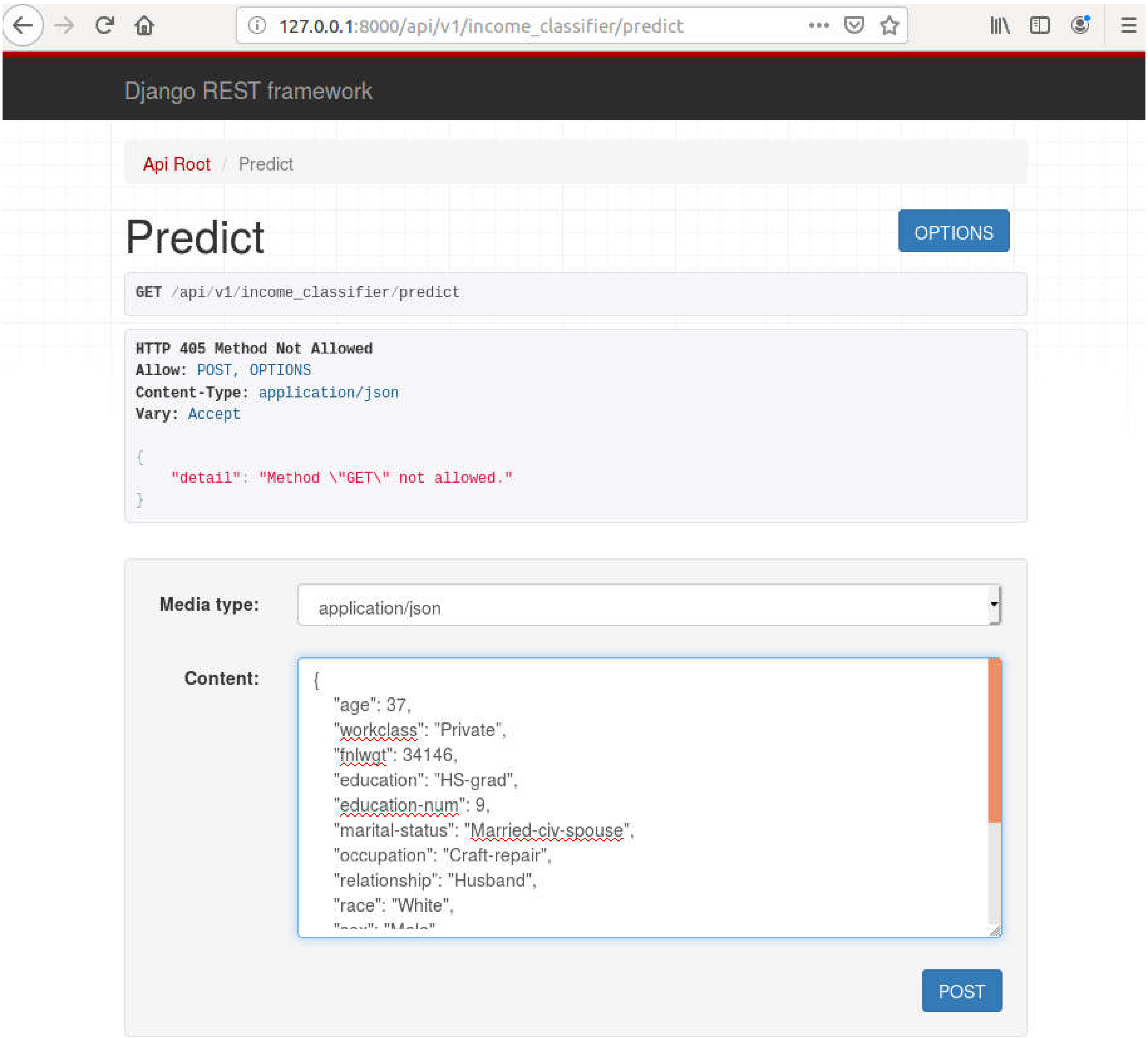
Based on endpoint name, status and version there is routing of the request to correct ML algorithm. If the algorithm is selected properly, the JSON request is forwarded to the algorithm object and prediction is computed.

In the code there is also included code that is drawing algorithm in case A/B testing, we will go into details of this code in the next chapter.

To check if is it working please go to `http://127.0.0.1:8000/api/v1/income_classifier/predict` and provide example JSON input:

```
{
    "age": 37,
    "workclass": "Private",
    "fnlwgt": 34146,
    "education": "HS-grad",
    "education-num": 9,
    "marital-status": "Married-civ-spouse",
    "occupation": "Craft-repair",
    "relationship": "Husband",
    "race": "White",
    "sex": "Male",
    "capital-gain": 0,
    "capital-loss": 0,
    "hours-per-week": 68,
    "native-country": "United-States"
}
```

and click the `POST` button. You should see views like in images 8 and 9.



Predict

```
GET /api/v1/income_classifier/predict
```

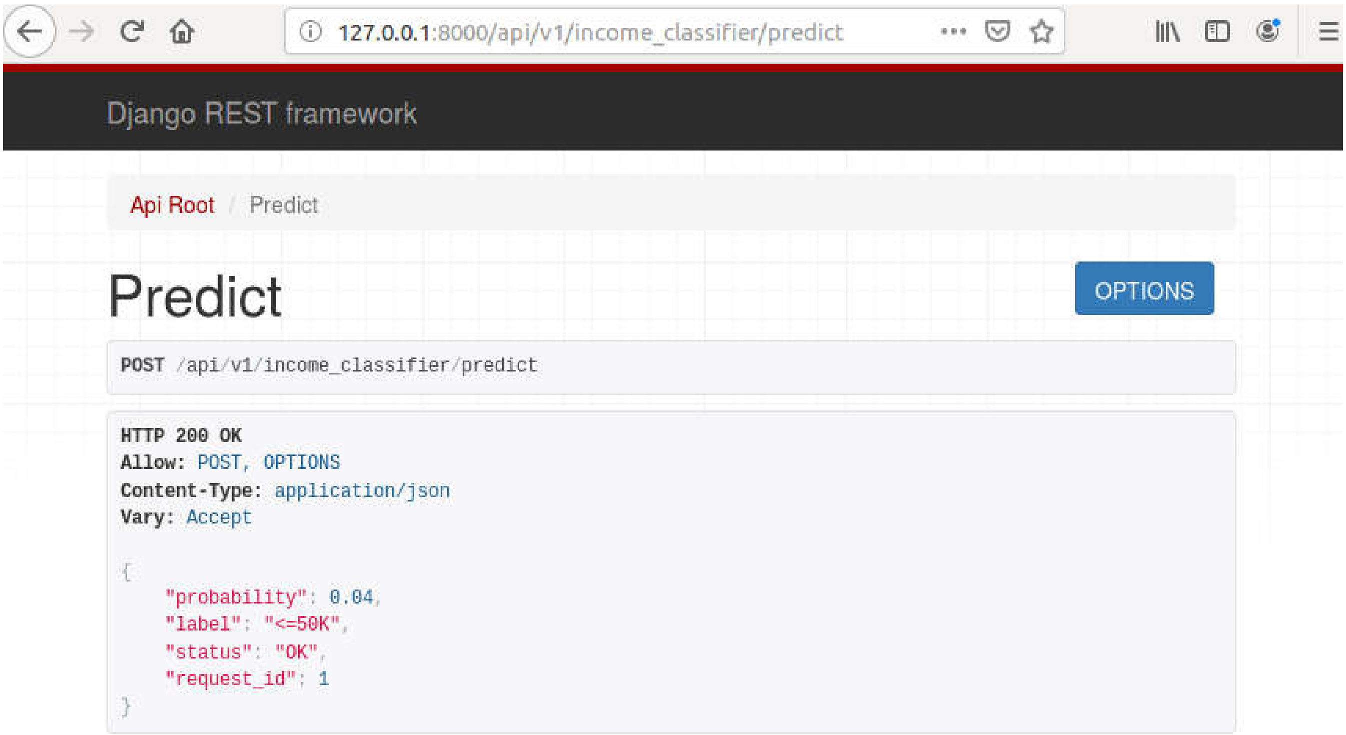
```
HTTP 405 Method Not Allowed
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "detail": "Method \"GET\" not allowed."
}
```

Media type: application/json

Content:

```
{
    "age": 37,
    "workclass": "Private",
    "fnlwgt": 34146,
    "education": "HS-grad",
    "education-num": 9,
    "marital-status": "Married-civ-spouse",
    "occupation": "Craft-repair",
    "relationship": "Husband",
    "race": "White",
```

Figure 8: Fill input data and click POST



```
POST /api/v1/income_classifier/predict
```

```
HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "probability": 0.04,
    "label": "<=50K",
    "status": "OK",
    "request_id": 1
}
```

Figure 9: Response of the ML algorithm

Congratulations!!! If you see the result as on image 9 it means that your ML web service is working correctly.

Your response should look like this:

```
{
    "probability": 0.04,
    "label": "<=50K",
    "status": "OK",
    "request_id": 1
}
```

The response contains `probability`, `label`, `status` and `request_id`. The `request_id` can be used later to provide feedback and ML algorithms monitoring.

Add tests for PredictView

We will add a simple test case that will check if the predicted view correctly responds to correct data.

```
# file backend/server/endpoints/tests.py
from django.test import TestCase
from rest_framework.test import APIClient

class EndpointTests(TestCase):

    def test_predict_view(self):
        client = APIClient()
        input_data = {
            "age": 37,
            "workclass": "Private",
            "fnlwgt": 34146,
            "education": "HS-grad",
            "education-num": 9,
            "marital-status": "Married-civ-spouse",
            "occupation": "Craft-repair",
            "relationship": "Husband",
            "race": "White",
            "sex": "Male",
            "capital-gain": 0,
            "capital-loss": 0,
            "hours-per-week": 68,
            "native-country": "United-States"
        }
        classifier_url = "/api/v1/income_classifier/predict"
        response = client.post(classifier_url, input_data, format='json')
        self.assertEqual(response.status_code, 200)
        self.assertEqual(response.data["label"], "<=50K")
        self.assertTrue("request_id" in response.data)
        self.assertTrue("status" in response.data)
```

To run this test:

```
# please run in backend/server directory
python manage.py test apps.endpoints.tests
```

To run all tests:

```
# please run in backend/server directory
python manage.py test apps
```

Later more tests can be added, which will cover situations, where wrong endpoints are selected in the URL or data, is in the wrong format.

Add code to the repository

Before going to next chapter let's add code to the repository:

```
git commit -am "add predict view"
git push
```

In the next chapter, we will work on the A/B testing of ML algorithms.

A/B testing

What you already did:

- create ML algorithms,
- create Django web service, with ML code, database models for endpoints, algorithms, and requests.
- create predict view, which is routing requests to ML algorithms.

What you will learn in this chapter:

- add a second ML algorithm (Extra Trees based) to the web service,
- create database model and REST API view for A/B tests information,
- write a python script for sending requests.

Add second ML algorithm

We will add code and tests for the Extra Trees based algorithm. Please add new file `extra_trees.py` in `backend/server/apps/ml/income_classifer` directory. (The code is very similar to `RandomForestClassifier` class but to keep it simple I just copy it and change the path for reading the model. There can be used inheritance here.).

```
# file backend/server/apps/ml/income_classifier/extra_trees.py
import joblib
import pandas as pd

class ExtraTreesClassifier:
    def __init__(self):
        path_to_artifacts = "../../research/"
        self.values_fill_missing =  joblib.load(path_to_artifacts + "train_mode.job
        self.encoders = joblib.load(path_to_artifacts + "encoders.joblib")
        self.model = joblib.load(path_to_artifacts + "extra_trees.joblib")

    def preprocessing(self, input_data):
        # JSON to pandas DataFrame
        input_data = pd.DataFrame(input_data, index=[0])
        # fill missing values
        input_data.fillna(self.values_fill_missing)
        # convert categoricals
        for column in [
            "workclass",
            "education",
            "marital-status",
            "occupation",
            "relationship",
            "race",
            "sex",
            "native-country",
        ]:
            categorical_convert = self.encoders[column]
            input_data[column] = categorical_convert.transform(input_data[column])

        return input_data

    def predict(self, input_data):
        return self.model.predict_proba(input_data)

    def postprocessing(self, input_data):
        label = "<=50K"
        if input_data[1] > 0.5:
            label = ">50K"
        return {"probability": input_data[1], "label": label, "status": "OK"}

    def compute_prediction(self, input_data):
        try:
            input_data = self.preprocessing(input_data)
            prediction = self.predict(input_data)[0]  # only one sample
            prediction = self.postprocessing(prediction)
        except Exception as e:
            return {"status": "Error", "message": str(e)}

        return prediction
```

Add the test in `backend/server/apps/ml/tests.py` file:

```
# in file backend/server/apps/ml/tests.py
# add new import
from apps.ml.income_classifier.extra_trees import ExtraTreesClassifier

# ... the rest of the code

# add new test method to MLTests class
    def test_et_algorithm(self):
        input_data = {
            "age": 37,
            "workclass": "Private",
            "fnlwgt": 34146,
            "education": "HS-grad",
            "education-num": 9,
            "marital-status": "Married-civ-spouse",
            "occupation": "Craft-repair",
            "relationship": "Husband",
            "race": "White",
            "sex": "Male",
            "capital-gain": 0,
            "capital-loss": 0,
            "hours-per-week": 68,
            "native-country": "United-States"
        }
        my_alg = ExtraTreesClassifier()
        response = my_alg.compute_prediction(input_data)
        self.assertEqual('OK', response['status'])
        self.assertTrue('label' in response)
        self.assertEqual('<=50K', response['label'])
```

To run tests:

```
# please run in backend/server directory
python manage.py test apps.ml.tests
```

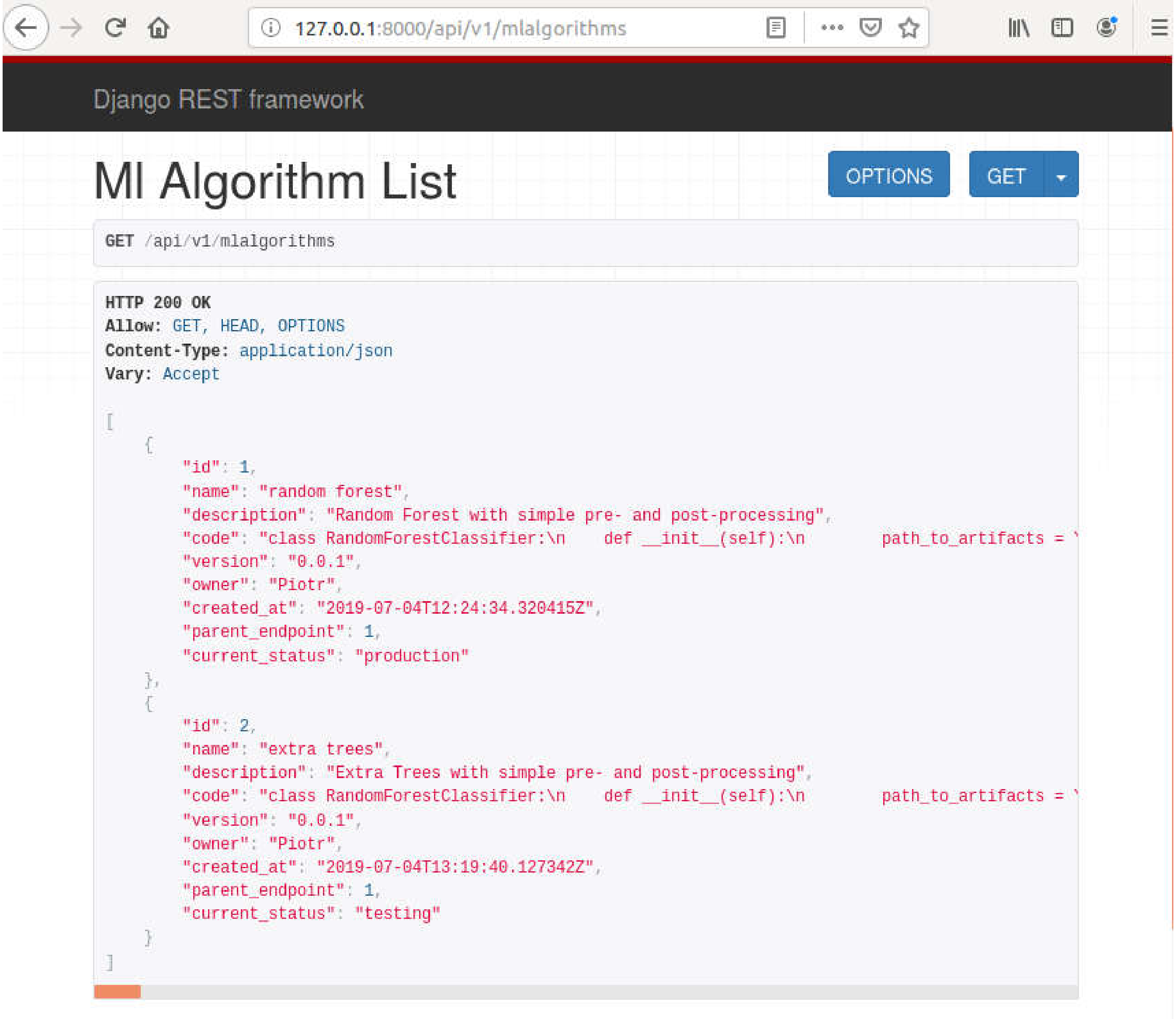
The algorithm is working as expected. We need to add it to our ML registry. We need to modify `backend/server/server/wsgi.py` file:

```
# the `backend/server/server/wsgi.py file
import os
from django.core.wsgi import get_wsgi_application
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'server.settings')
application = get_wsgi_application()

# ML registry
import inspect
from apps.ml.registry import MLRegistry
from apps.ml.income_classifier.random_forest import RandomForestClassifier
from apps.ml.income_classifier.extra_trees import ExtraTreesClassifier # import Ext

try:
    registry = MLRegistry() # create ML registry
    # Random Forest classifier
    rf = RandomForestClassifier()
    # add to ML registry
    registry.add_algorithm(endpoint_name="income_classifier",
                            algorithm_object=rf,
                            algorithm_name="random forest",
                            algorithm_status="production",
                            algorithm_version="0.0.1",
                            owner="Piotr",
                            algorithm_description="Random Forest with simple pre- a
                            algorithm_code=inspect.getsource(RandomForestClassifier

    # Extra Trees classifier
    et = ExtraTreesClassifier()
    # add to ML registry
    registry.add_algorithm(endpoint_name="income_classifier",
                            algorithm_object=et,
                            algorithm_name="extra trees",
                            algorithm_status="testing",
                            algorithm_version="0.0.1",
                            owner="Piotr",
                            algorithm_description="Extra Trees with simple pre- and
                            algorithm_code=inspect.getsource(RandomForestClassifier
except Exception as e:
    print("Exception while loading the algorithms to the registry,", str(e))
```

To see changes, please restart the server:

```
# please run in backend/server
# stop server with CONTROL-C.
# start server:
python manage.py runserver
```

After server restart please open `http://127.0.0.1:8000/api/v1/mlalgorithms` in the web browser. You should see two registered ML algorithms (image 10).



Ml Algorithm List

GET /api/v1/mlalgorithms

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "random forest",
        "description": "Random Forest with simple pre- and post-processing",
        "code": "class RandomForestClassifier:\n    def __init__(self):\n        path_to_artifacts = '
        "version": "0.0.1",
        "owner": "Piotr",
        "created_at": "2019-07-04T12:24:34.320415Z",
        "parent_endpoint": 1,
        "current_status": "production"
    },
    {
        "id": 2,
        "name": "extra trees",
        "description": "Extra Trees with simple pre- and post-processing",
        "code": "class RandomForestClassifier:\n    def __init__(self):\n        path_to_artifacts = '
        "version": "0.0.1",
        "owner": "Piotr",
        "created_at": "2019-07-04T13:19:40.127342Z",
        "parent_endpoint": 1,
        "current_status": "testing"
    }
]
```

Figure 10: Two ML algorithms registered in the service

Create A/B model in the database

Add ABTest model

Let's add database model in the `backend/server/apps/endpoints/models.py` file to keep information about A/B tests:

```
# please add at the end of file backend/server/apps/endpoints/models.py

class ABTest(models.Model):
    '''
    The ABTest will keep information about A/B tests.
    Attributes:
        title: The title of test.
        created_by: The name of creator.
        created_at: The date of test creation.
        ended_at: The date of test stop.
        summary: The description with test summary, created at test stop.
        parent_mlalgorithm_1: The reference to the first corresponding MLAlgorithm.
        parent_mlalgorithm_2: The reference to the second corresponding MLAlgorithm
    '''
    title = models.CharField(max_length=10000)
    created_by = models.CharField(max_length=128)
    created_at = models.DateTimeField(auto_now_add=True, blank=True)
    ended_at = models.DateTimeField(blank=True, null=True)
    summary = models.CharField(max_length=10000, blank=True, null=True)

    parent_mlalgorithm_1 = models.ForeignKey(MLAlgorithm, on_delete=models.CASCADE,
    parent_mlalgorithm_2 = models.ForeignKey(MLAlgorithm, on_delete=models.CASCADE,
```

The `ABTest` keeps information about:

- which ML algorithms are tested,
- who and when created the test,
- when test is stopped,
- the test results in the `summary` field.

Define serializer

Let's add a serializer for the `ABTest` model.

```
# please add at the beginning of file backend/server/apps/endpoints/serializers.py

from apps.endpoints.models import ABTest

# ...
# rest of the code
# ...

# please add at the end of file backend/server/apps/endpoints/serializers.py
class ABTestSerializer(serializers.ModelSerializer):
    class Meta:
        model = ABTest
        read_only_fields = (
            "id",
            "ended_at",
            "created_at",
            "summary",
        )
        fields = (
            "id",
            "title",
            "created_by",
            "created_at",
            "ended_at",
            "summary",
            "parent_mlalgorithm_1",
            "parent_mlalgorithm_2",
            )
```

Please notice, that `id`, `created_at`, `ended_at` and `summary` fields are marked as read-only. We will allow users to create A/B tests with REST API the read-only fields with be set with server code.

Define view

```
# please add to the file backend/server/apps/endpoints/views.py

from django.db import transaction
from apps.endpoints.models import ABTest
from apps.endpoints.serializers import ABTestSerializer


class ABTestViewSet(
    mixins.RetrieveModelMixin, mixins.ListModelMixin, viewsets.GenericViewSet,
    mixins.CreateModelMixin, mixins.UpdateModelMixin
):
    serializer_class = ABTestSerializer
    queryset = ABTest.objects.all()

    def perform_create(self, serializer):
        try:
            with transaction.atomic():
                instance = serializer.save()
                # update status for first algorithm

                status_1 = MLAlgorithmStatus(status = "ab_testing",
                                created_by=instance.created_by,
                                parent_mlalgorithm = instance.parent_mlalgorithm_1,
                                active=True)
                status_1.save()
                deactivate_other_statuses(status_1)
                # update status for second algorithm
                status_2 = MLAlgorithmStatus(status = "ab_testing",
                                created_by=instance.created_by,
                                parent_mlalgorithm = instance.parent_mlalgorithm_2,
                                active=True)
                status_2.save()
                deactivate_other_statuses(status_2)

        except Exception as e:
            raise APIException(str(e))
```

The `ABTestViewSet` view allows the user to create new objects. The `perform_create` method creates the `ABTest` object and two new statuses for ML algorithms. The new statuses are set to `ab_testing`.

We will add also a view to stop the A/B test.

```
# please add to the file backend/server/apps/endpoints/views.py

from django.db.models import F
import datetime

class StopABTestView(views.APIView):
    def post(self, request, ab_test_id, format=None):

        try:
            ab_test = ABTest.objects.get(pk=ab_test_id)

            if ab_test.ended_at is not None:
                return Response({"message": "AB Test already finished."})

            date_now = datetime.datetime.now()
            # alg #1 accuracy
            all_responses_1 = MLRequest.objects.filter(parent_mlalgorithm=ab_test.p
            correct_responses_1 = MLRequest.objects.filter(parent_mlalgorithm=ab_te
            accuracy_1 = correct_responses_1 / float(all_responses_1)
            print(all_responses_1, correct_responses_1, accuracy_1)

            # alg #2 accuracy
            all_responses_2 = MLRequest.objects.filter(parent_mlalgorithm=ab_test.p
            correct_responses_2 = MLRequest.objects.filter(parent_mlalgorithm=ab_te
            accuracy_2 = correct_responses_2 / float(all_responses_2)
            print(all_responses_2, correct_responses_2, accuracy_2)

            # select algorithm with higher accuracy
            alg_id_1, alg_id_2 = ab_test.parent_mlalgorithm_1, ab_test.parent_mlalg
            # swap
            if accuracy_1 < accuracy_2:
                alg_id_1, alg_id_2 = alg_id_2, alg_id_1

            status_1 = MLAlgorithmStatus(status = "production",
                            created_by=ab_test.created_by,
                            parent_mlalgorithm = alg_id_1,
                            active=True)
            status_1.save()
            deactivate_other_statuses(status_1)
            # update status for second algorithm
            status_2 = MLAlgorithmStatus(status = "testing",
                            created_by=ab_test.created_by,
                            parent_mlalgorithm = alg_id_2,
                            active=True)
            status_2.save()
            deactivate_other_statuses(status_2)


            summary = "Algorithm #1 accuracy: {}, Algorithm #2 accuracy: {}".format
            ab_test.ended_at = date_now
            ab_test.summary = summary
            ab_test.save()

        except Exception as e:
            return Response({"status": "Error", "message": str(e)},
                            status=status.HTTP_400_BAD_REQUEST
            )
        return Response({"message": "AB Test finished.", "summary": summary})
```

The `StopABTestView` stops the A/B test and compute the accuracy (ratio of correct responses) for each algorithm. The algorithm with higher accurcy is set as production algorithm, the other algorithm is saved with `testing` status.

Add URL router for ABTest

The last thing is to add the URL router:

```
# the backend/server/apps/endpoints/urls.py file
from django.conf.urls import url, include
from rest_framework.routers import DefaultRouter

from apps.endpoints.views import EndpointViewSet
from apps.endpoints.views import MLAlgorithmViewSet
from apps.endpoints.views import MLAlgorithmStatusViewSet
from apps.endpoints.views import MLRequestViewSet
from apps.endpoints.views import PredictView
from apps.endpoints.views import ABTestViewSet
from apps.endpoints.views import StopABTestView

router = DefaultRouter(trailing_slash=False)
router.register(r"endpoints", EndpointViewSet, basename="endpoints")
router.register(r"mlalgorithms", MLAlgorithmViewSet, basename="mlalgorithms")
router.register(r"mlalgorithmstatuses", MLAlgorithmStatusViewSet, basename="mlalgor
router.register(r"mlrequests", MLRequestViewSet, basename="mlrequests")
router.register(r"abtests", ABTestViewSet, basename="abtests")

urlpatterns = [
    url(r"^api/v1/", include(router.urls)),
    url(
        r"^api/v1/(?P<endpoint_name>.+)/predict$", PredictView.as_view(), name="pre
    ),
    url(
        r"^api/v1/stop_ab_test/(?P<ab_test_id>.+)", StopABTestView.as_view(), name=
    ),
]
```

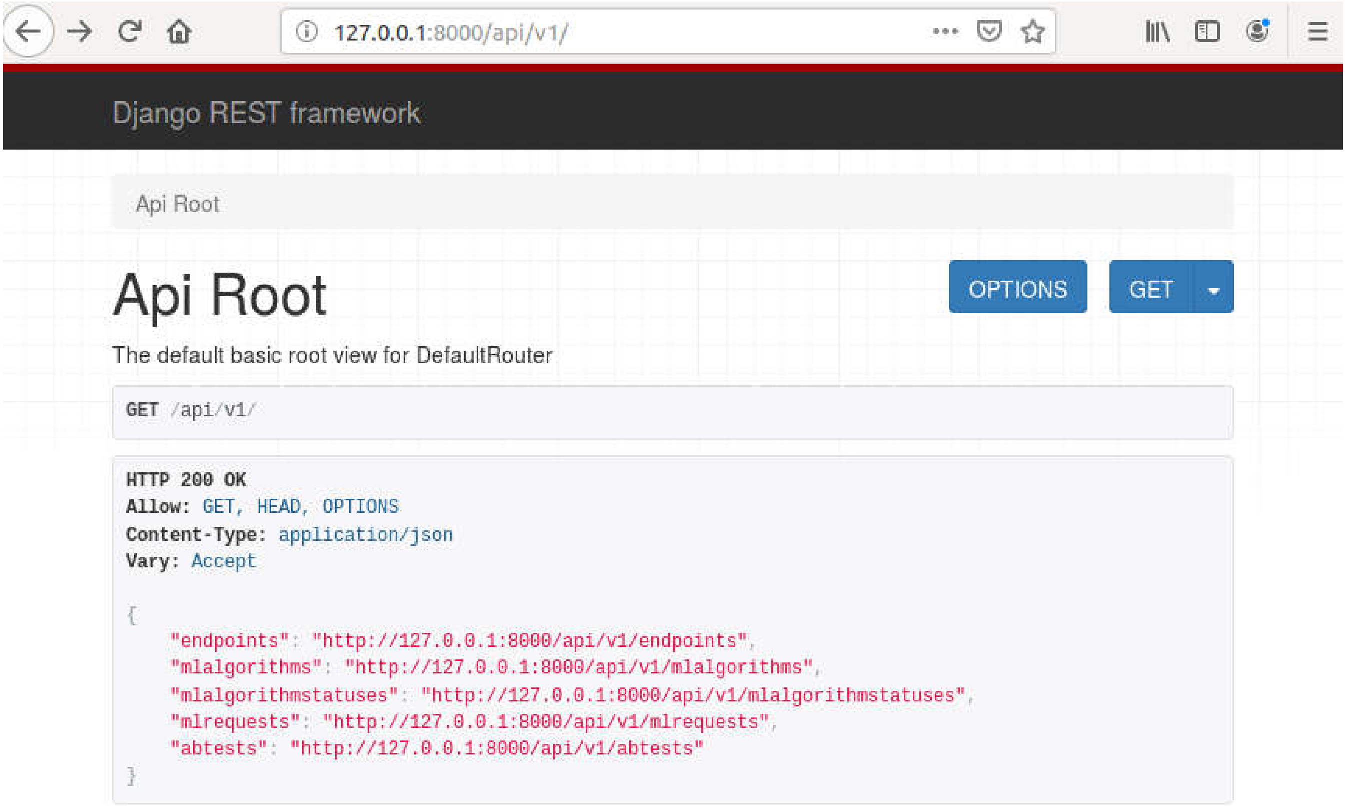
OK, we are almost set. Before starting a development server we need to create and apply database migrations:

```
python manage.py makemigrations
python manage.py migrate
```

Let's run the server:

```
# please run in backend/server
python manage.py runserver
```

You should see list of DRF generated list of APIs like in image 11.



Api Root

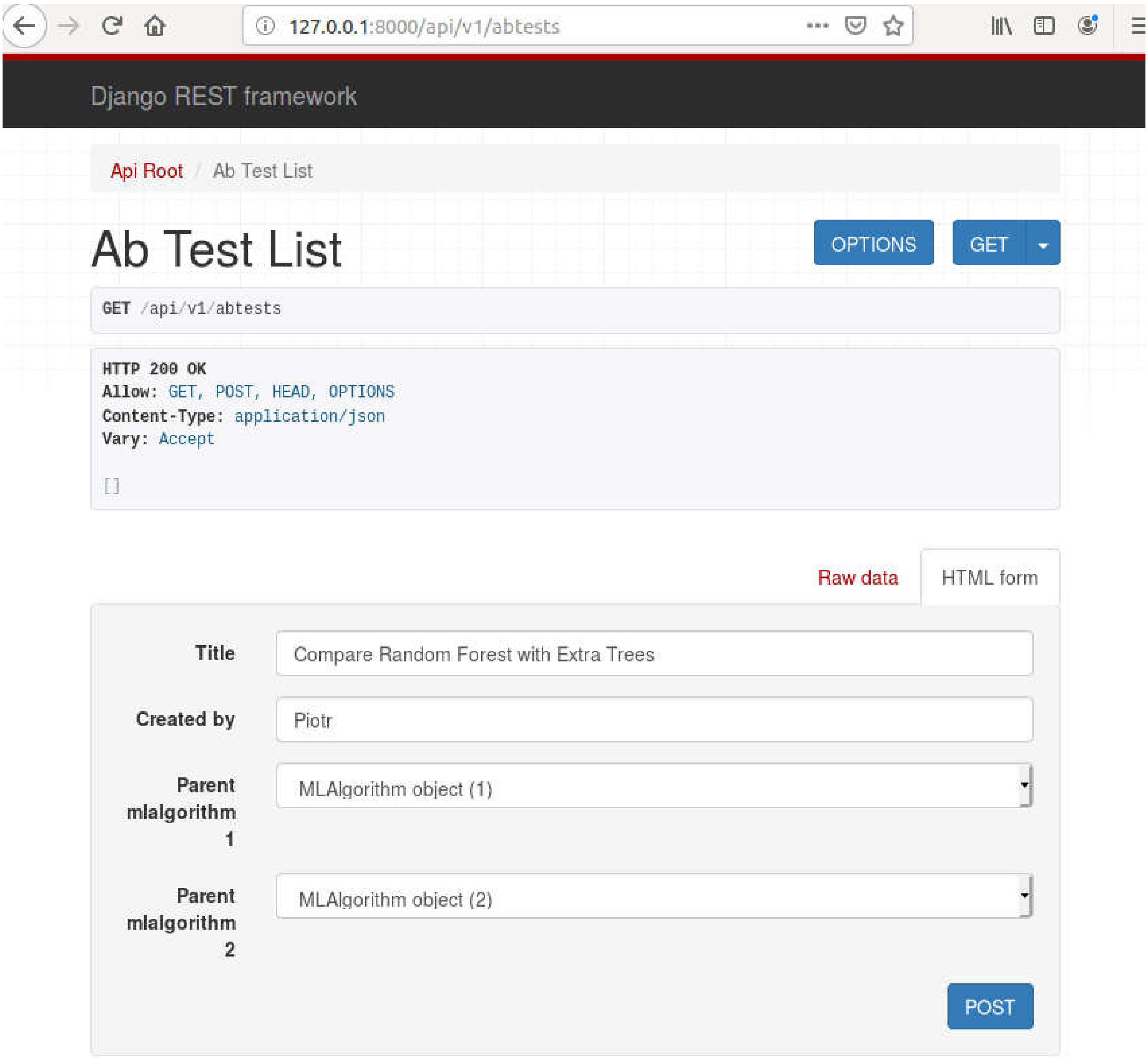
The default basic root view for DefaultRouter

```
GET /api/v1/
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "endpoints": "http://127.0.0.1:8000/api/v1/endpoints",
    "mlalgorithms": "http://127.0.0.1:8000/api/v1/mlalgorithms",
    "mlalgorithmstatuses": "http://127.0.0.1:8000/api/v1/mlalgorithmstatuses",
    "mlrequests": "http://127.0.0.1:8000/api/v1/mlrequests",
    "abtests": "http://127.0.0.1:8000/api/v1/abtests"
}
```

Figure 11: URL to A/B tests

Let's start new A/B test. Please go to address `http://127.0.0.1:8000/api/v1/abtests` (at development environment). Please set the title, creator name and set algorithms. You have algorithm id in the brackets. Make sure that you select id 1 and 2, like in the image 12. Press the POST button to create the test.



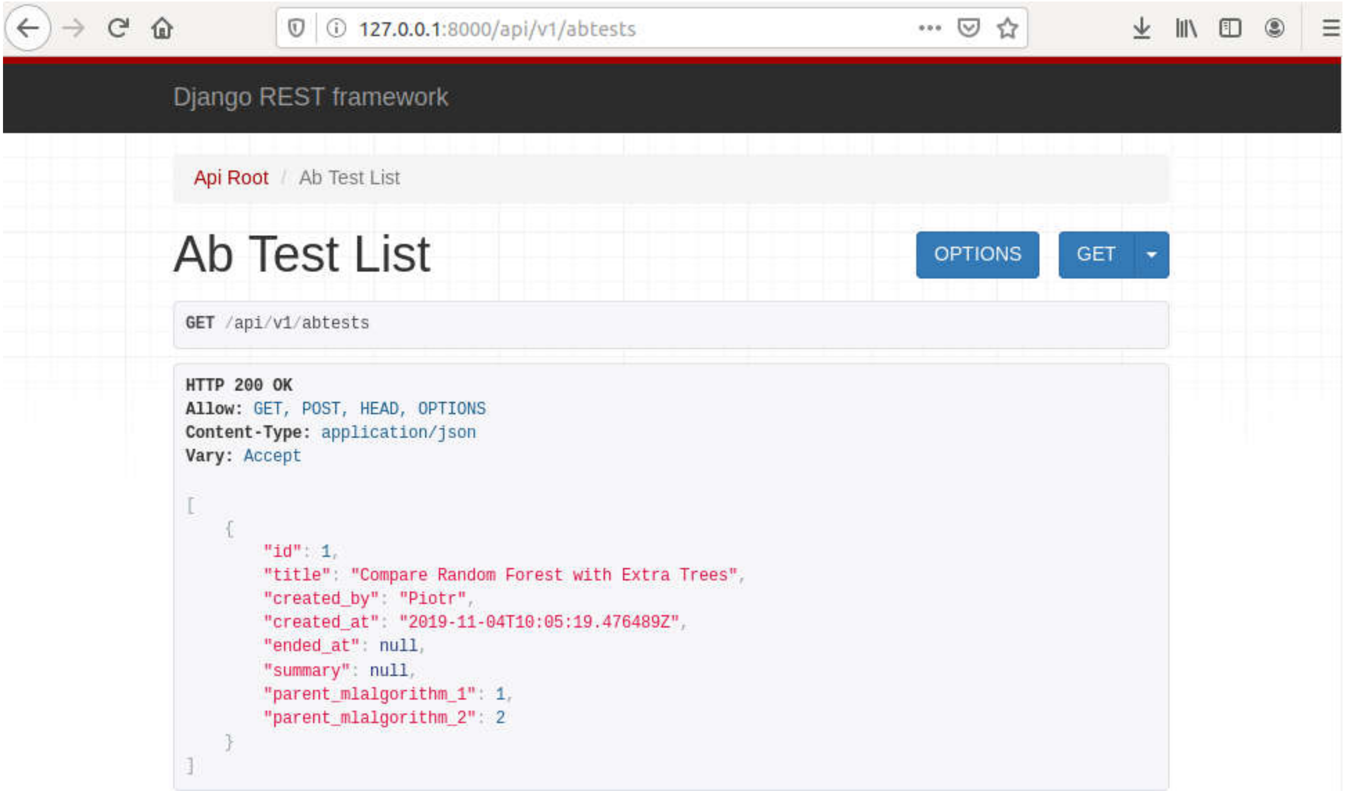
```
GET /api/v1/abtests
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[]
```

Title	Compare Random Forest with Extra Trees
Created by	Piotr
Parent mlalgorithm 1	MLAlgorithm object (1)
Parent mlalgorithm 2	MLAlgorithm object (2)

Figure 12: View to create new A/B test

After new A/B test creation you should see view like in the image 13.

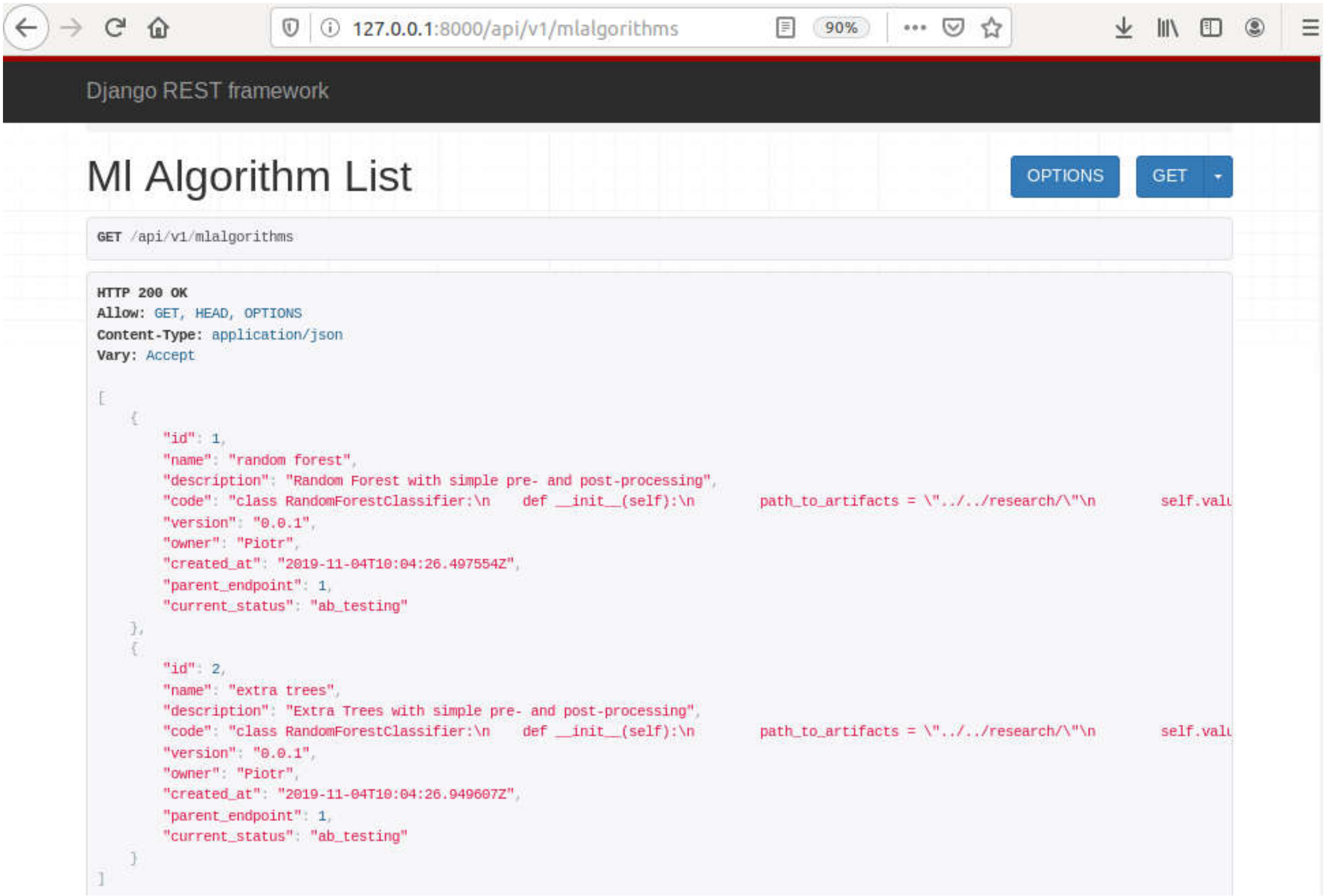


```
GET /api/v1/abtests
```

```
HTTP 200 OK
Allow: GET, POST, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "title": "Compare Random Forest with Extra Trees",
        "created_by": "Piotr",
        "created_at": "2019-11-04T10:05:19.476489Z",
        "ended_at": null,
        "summary": null,
        "parent_mlalgorithm_1": 1,
        "parent_mlalgorithm_2": 2
    }
]
```

Figure 13: Created new A/B test

After A/B test creation you should see updated status fields for ML algorithms. They should be set to `ab_testing`, like in the image 16.



```
GET /api/v1/mlalgorithms
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "name": "random forest",
        "description": "Random Forest with simple pre- and post-processing",
        "code": "class RandomForestClassifier:\n    def __init__(self):\n        path_to_artifacts = \"../../research/\"\n        self.valu
        "version": "0.0.1",
        "owner": "Piotr",
        "created_at": "2019-11-04T10:04:26.497554Z",
        "parent_endpoint": 1,
        "current_status": "ab_testing"
    },
    {
        "id": 2,
        "name": "extra trees",
        "description": "Extra Trees with simple pre- and post-processing",
        "code": "class RandomForestClassifier:\n    def __init__(self):\n        path_to_artifacts = \"../../research/\"\n        self.valu
        "version": "0.0.1",
        "owner": "Piotr",
        "created_at": "2019-11-04T10:04:26.949607Z",
        "parent_endpoint": 1,
        "current_status": "ab_testing"
    }
]
```

Figure 14: ML algorithms with updates statuses

Run the A/B test

To run the A/B test we will write python script in the Jupyter notebook that will simulate real life A/B testing. The script will:

- read test data,
- send sample by sample to the server,
- get the server response and send the feedback to the server.

Before starting new notebook, please install `requests` package that will be used for communication with the server.

```
pip3 install requests
```

Please open Jupyter notebook and create new script `ab_test.ipynb` in the `research` directory.

Let's add necessary packages.

```
import json # will be needed for saving preprocessing details
import numpy as np # for data manipulation
import pandas as pd # for data manipulation
from sklearn.model_selection import train_test_split # will be used for data split
import requests
```

Code to read the data:

```
# load dataset
df = pd.read_csv('https://raw.githubusercontent.com/pplonski/datasets-for-start/mas
x_cols = [c for c in df.columns if c != 'income']
# set input matrix and target column
X = df[x_cols]
y = df['income']
# show first rows of data
df.head(
```

Split the data to train and test sets:

```
# data split train / test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3, random_s
```

Please notice that we used the same seed (`random_state` value) as earlier while model training.

Let's use first 100 rows of test data for A/B test.

```
for i in range(100):
    input_data = dict(X_test.iloc[i])
    target = y_test.iloc[i]
    r = requests.post("http://127.0.0.1:8000/api/v1/income_classifier/predict?statu
    response = r.json()
    # provide feedback
    requests.put("http://127.0.0.1:8000/api/v1/mlrequests/{}".format(response["requ
```

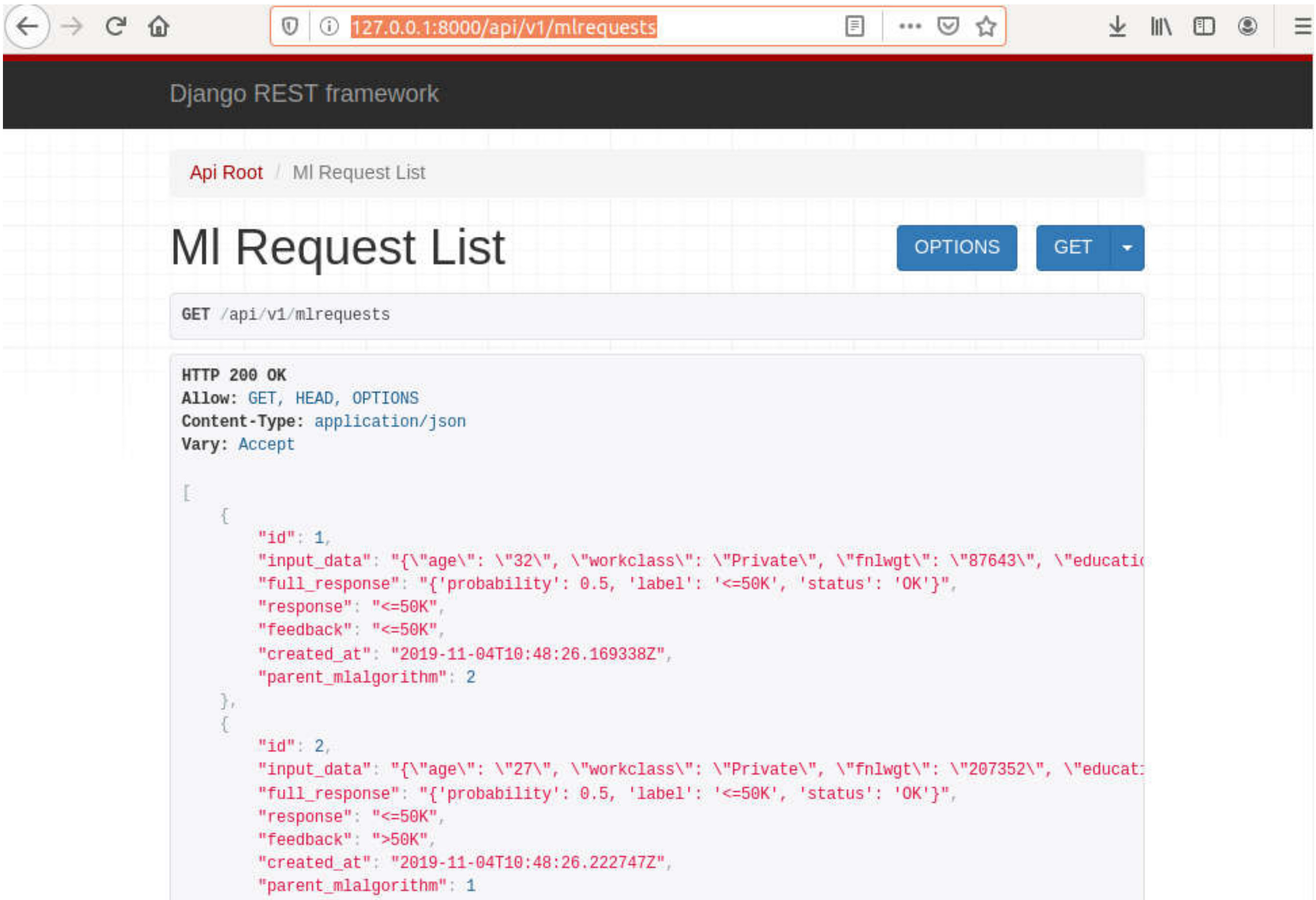
In each iteration step, we are sending data to API endpoint:

```
http://127.0.0.1:8000/api/v1/income_classifier/predict?status=ab_testing
```

and provide feedback with true label at:

```
http://127.0.0.1:8000/api/v1/mlrequests/<request-id>
```

After running the script, you can check the requests at address: `http://127.0.0.1:8000/api/v1/mlrequests`. You should see list of requests like in the image 15.



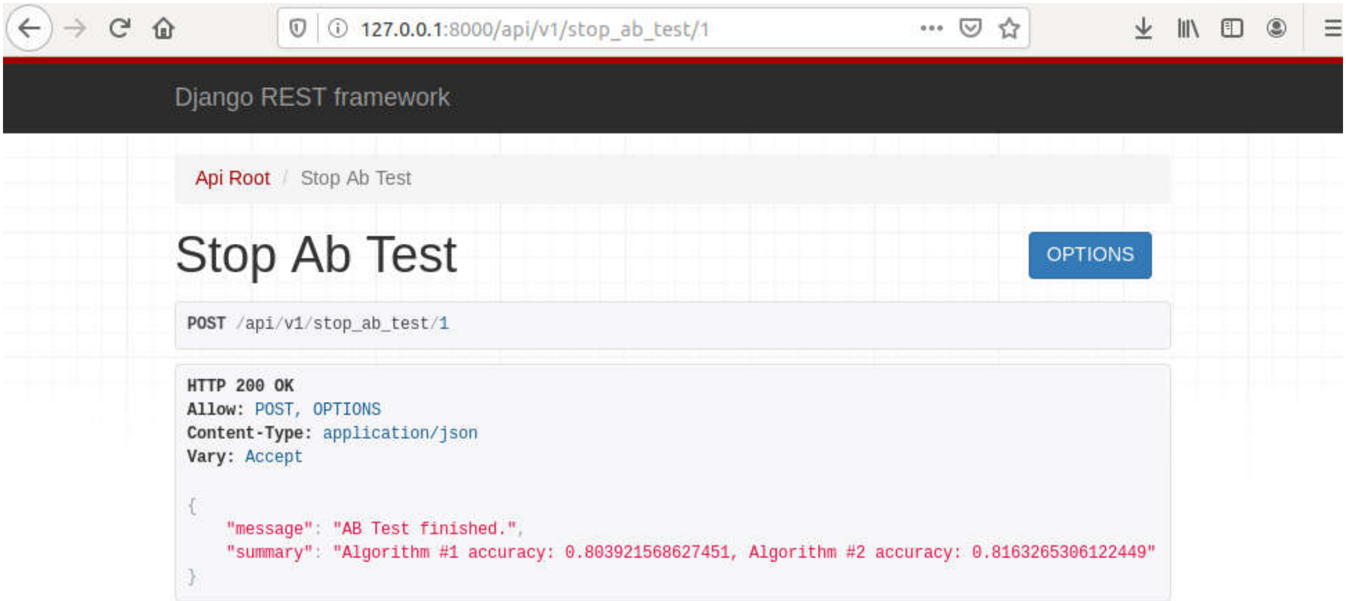
Ml Request List

```
GET /api/v1/mlrequests
```

```
HTTP 200 OK
Allow: GET, HEAD, OPTIONS
Content-Type: application/json
Vary: Accept

[
    {
        "id": 1,
        "input_data": "{\"age\": \"32\", \"workclass\": \"Private\", \"fnlwgt\": \"87643\", \"educatic
        "full_response": "{'probability': 0.5, 'label': '<=50K', 'status': 'OK'}",
        "response": "<=50K",
        "feedback": "<=50K",
        "created_at": "2019-11-04T10:48:26.169338Z",
        "parent_mlalgorithm": 2
    },
    {
        "id": 2,
        "input_data": "{\"age\": \"27\", \"workclass\": \"Private\", \"fnlwgt\": \"207352\", \"educat:
        "full_response": "{'probability': 0.5, 'label': '<=50K', 'status': 'OK'}",
        "response": "<=50K",
        "feedback": ">50K",
        "created_at": "2019-11-04T10:48:26.222747Z",
        "parent_mlalgorithm": 1
```

Figure 15: ML requests after running the A/B test script

To stop the A/B test, please open address `http://127.0.0.1:8000/api/v1/stop_ab_test/1` where `1` at the end of the address it the A/B test id. Click on POST button to finish A/B test. You should get the view like in the image @fig:16.



```
POST /api/v1/stop_ab_test/1
```

```
HTTP 200 OK
Allow: POST, OPTIONS
Content-Type: application/json
Vary: Accept

{
    "message": "AB Test finished.",
    "summary": "Algorithm #1 accuracy: 0.803921568627451, Algorithm #2 accuracy: 0.8163265306122449"
}
```

Figure 16: A/B test finish

You can see that there is summary of the test displayed with accuracy for each algorithm. You can check (at `http://127.0.0.1:8000/api/v1/mlalgorithms`) that algorithms have updated statuses, and the model with higher accuracy is set to production.

Add code to the repository

Let's save our code to the repository:

```
git add backend/server/apps/ml/income_classifier/extra_trees.py
git add research/ab_test.ipynb
git commit -am "ab tests"
git push
```

In the next chapter, we will define docker container for our server.

Containers

What you already did:

- create ML algorithms,

- create Django web service, with ML code, database models for endpoints, algorithms, and requests,
- create predict view, which is routing requests to ML algorithms,
- create A/B testing code in the server.

In this chapter you will define docker container for our server code. With docker it is easy to deploy the code to selected infrastructure and it is easier to scale the service if needed.

Prepare the code

Before creating the docker definition we need to add some changes in the server code.

Please edit `backend/server/server/settings.py` file and set `ALLOWED_HOSTS` variable:

```
ALLOWED_HOSTS = ['0.0.0.0']
```

Additionally, set the `STATIC_ROOT`, `STATIC_URL` variables and the end of settings:

```
STATIC_ROOT = os.path.join(BASE_DIR, 'static')
STATIC_URL = '/static/'
```

Please add the `requirements.txt` file in the project's main directory:

```
Django==2.2.4
django-filter==2.2.0
djangorestframework==3.10.3
joblib==0.14.0
Markdown==3.1.1
numpy==1.17.3
pandas==0.25.2
requests==2.22.0
scikit-learn==0.21.3
```

Dockerfiles

Let's define the docker files for nginx server and our server application. We will keep them in separate directories:

```
# please run in project's main directory
mkdir docker
mkdir docker/nginx
mkdir docker/backend
```

Please add file `Dockerfile` in `docker/nginx` directory:

```
# docker/nginx/Dockerfile
FROM nginx:1.13.12-alpine
CMD ["nginx", "-g", "daemon off;"]
```

Additionally, we will add nginx config file, please add `docker/nginx/default.conf` file:

```
server {
    listen 8000 default_server;
    listen [::]:8000;

    client_max_body_size 20M;

    location / {
        try_files $uri @proxy_api;
    }

    location @proxy_api {
        proxy_set_header X-Forwarded-Proto https;
        proxy_set_header X-Url-Scheme $scheme;
        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header Host $http_host;
        proxy_redirect off;
        proxy_pass   http://wsgiserver:8000;
    }

    location /static/ {
        autoindex on;
        alias /app/backend/server/static/;
    }

}
```

Now, let's define 'Dockerfile' for our server application. Please add file `docker/backend/Dockerfile`:

```
FROM ubuntu:xenial

RUN apt-get update && \
    apt-get install -y software-properties-common && \
    add-apt-repository ppa:deadsnakes/ppa && \
    apt-get update && \
    apt-get install -y python3.6 python3.6-dev python3-pip

WORKDIR /app
COPY requirements.txt .
RUN rm -f /usr/bin/python && ln -s /usr/bin/python3.6 /usr/bin/python
RUN rm -f /usr/bin/python3 && ln -s /usr/bin/python3.6 /usr/bin/python3

RUN pip3 install -r requirements.txt
RUN pip3 install gunicorn==19.9.0

ADD ./backend /app/backend
ADD ./docker /app/docker
ADD ./research /app/research

RUN mkdir -p /app/backend/server/static
```

In this dockerfile, we load ubuntu system, and install all needed packages and switch default python to python 3.6. At the end, we copy the application code.

We will define starting script for our application. Please add `docker/backend/wsgi-entrypoint.sh` file:

```
#!/usr/bin/env bash

echo "Start backend server"
until cd /app/backend/server
do
    echo "Waiting for server volume..."
done

until ./manage.py migrate
do
    echo "Waiting for database to be ready..."
    sleep 2
done

./manage.py collectstatic --noinput

gunicorn server.wsgi --bind 0.0.0.0:8000 --workers 4 --threads 4
```

We will use this starting script to apply database migrations and creation of static files before application is stated with gunicorn.

We have dockerfiles defined for nginx server and our application. We will manage them with `docker-compose` command. Let's add `docker-compose.yml` file in the main directory:

```
version: '2'

services:
    nginx:
        restart: always
        image: nginx:1.12-alpine
        ports:
            - 8000:8000
        volumes:
            - ./docker/nginx/default.conf:/etc/nginx/conf.d/default.conf
            - static_volume:/app/backend/server/static
    wsgiserver:
        build:
            context: .
            dockerfile: ./docker/backend/Dockerfile
        entrypoint: /app/docker/backend/wsgi-entrypoint.sh
        volumes:
            - static_volume:/app/backend/server/static
        expose:
            - 8000
volumes:
    static_volume: {}
```

To build docker images please run:

```
sudo docker-compose build
```

To start the docker images please run:

```
sudo docker-compose up
```

You should be able to see the running server at the address:

```
http://0.0.0.0:8000/api/v1/
```

Congratulations!

That was the last step of this tutorial. You have successfully created your own web service that can serve machine learning models. Congratulations!

The full code is available in github https://github.com/pplonski/my_ml_service (https://github.com/pplonski/my_ml_service).

Feedback

I'm looking to you feedback!