

eBook

# Move your data science to production: Deploying dashboards & APIs

---

Go from model to production in minutes

**Stephanie Kirmer,**  
Senior Data Scientist,  
Saturn Cloud

**Free  
eBook**

Download now

# OVERVIEW

APIs and web applications are common methods of productionizing machine learning work. They allow users to supply inputs and swiftly receive inference results from your trained model, no matter where they are. In an API, programmatic calls can be used to retrieve JSON output, while web applications are more friendly to the business user and return attractive HTML output.

Training machine learning models is a core component of modern business. But taking those machine learning models and creating systems so that they continuously run, can often be an afterthought. These models can be deployed as dashboards for business stakeholders or APIs for engineers to call.

Saturn Cloud is designed to give data scientists what they need to be productive and generate value from machine learning. In this guide, we'll show you step-by-step, how to implement these two accessible techniques for productionizing models, and give you all the tools you need to make machine learning add value to your team and stakeholders.

# Part 1

# The Model

Data science model deployment can sound intimidating if you've never had a chance to try it in a safe space. Do you want to make a REST API, or a full frontend app? What does it take to do either of these? It's not as hard as you might think.

In this part, you'll learn how you can take a model and deploy it to a web app or a REST API (using Saturn Cloud), so that others can interact with it.

In this tool, we'll let the user make some feature selections, and then the model will predict an outcome for them. But using this same idea, you could easily do other things, such as letting the user retrain the model, upload things like images, or conduct other interactions with your model.

Just to be interesting, we're going to do this same project with two frameworks, Voilà and Flask, so you can see how they both work and decide what's right for your needs. In Flask, we'll create both a REST API and a web app version.

## Our Toolkit

[Saturn Cloud](#)

[Flask](#)

[Plotly](#) (python and JS)

[Scikit-learn](#) (for our model)

## Other Helpful Links

[codebook for the dataset](#)

[plotly.js cheat sheet](#)

[Jinja](#) (helpful for Flask)

## ..... The Project

The first steps of our process are exactly the same, whether we are going for Voila or Flask. We need to get some data and build a model. We're taking the US Department of Education's College Scorecard data, and building a quick linear regression model that accepts a few inputs and predicts a student's likely earnings 2 years after graduation. (You can get this data yourself [here](#))

### About Measurements

According to the data codebook: "The cohort of evaluated graduates for earnings metrics consists of those individuals who received federal financial aid, but excludes those who were subsequently enrolled in school during the measurement year, died prior to the end of the measurement year, received a higher-level credential than the credential level of the field of study measured, or did not work during the measurement year."

```
from sklearn.model_selection import train_test_split
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
from sklearn.preprocessing import OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn import linear_model
import pandas as pd
import numpy as np
import s3fs
import plotly
import plotly.graph_objects as go

# Flask app imports
from flask import Flask, render_template, request, jsonify
import json

# Voila app imports
from IPython.core.display import display, HTML
import ipywidgets as widgets
from ipywidgets import GridBox, Layout
```

For the GitHub Gist, click [here](#).

## Load Data

I already did some data cleaning and uploaded the features I wanted to a public bucket on S3, for easy access. This way, I can load it quickly when the app is run.

```
def load_data():
    s3 = s3fs.S3FileSystem(anon=True)
    s3path = 's3://saturn-public-data/college-scorecard/cleaned_merged.csv'

    major = pd.read_csv(
        s3path,
        storage_options={'anon': True},
        dtype = 'object',
        na_values = ['PrivacySuppressed']
    )
    return major
```

For the GitHub Gist, click [here](#).

## Format for Training

Once we have the dataset, this is going to give us a handful of features and our outcome. We just need to split it between features and target with scikit-learn to be ready to model. (Note that all of these functions will be run exactly as written in each of our apps.)

```
def split_data(df):

    X = df[['SAT_AVG_ALL', 'CREDESC', 'CIPDESC_new', 'CONTROL',
            'REGION', 'tuition', 'LOCALE', 'ADM_RATE_ALL']]
    y = df[['EARN_MDN_HI_2YR']]

    return [X, y]
```

For the GitHub Gist, click [here](#).



Our features are:

- REGION: Geographic location of college
- LOCALE: Type of city or town the college is in
- CONTROL: Type of college (public/private/for-profit)
- CIPDESC\_new: Major field of study (CIP code)
- CREDESC: Credential (Bachelor, Master, etc)
- ADM\_RATE\_ALL: Admission rate
- SAT\_AVG\_ALL: Average SAT score for admitted students (proxy for college prestige)
- tuition: Cost to attend the institution for one year

Our target outcome is EARN\_MDN\_HI\_2YR: median earnings measured two years after completion of degree

## Train Model

We are going to use scikit-learn's pipeline to make our feature engineering as easy and quick as possible. We're going to return a trained model as well as the R-squared value for the test sample, so we have a quick and straightforward measure of the model's performance on the test set that we can return along with the model object.

```
def trainmodel(X, y):
    enc = OneHotEncoder(handle_unknown='ignore', sparse = False)
    imp = IterativeImputer(max_iter=10, random_state=0, initial_strategy='mean', add_indicator = True)

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.15, random_state=42)

    ct = ColumnTransformer(
        [('onehot', enc, ['CONTROL', 'CREDESC', 'CIPDESC_new', 'REGION', 'LOCALE']),
         ('impute', imp, ['SAT_AVG_ALL', 'ADM_RATE_ALL'])],
        remainder='passthrough'
    )

    pipe = Pipeline(steps=[('coltrans', ct), ('linear', linear_model.LinearRegression())])
    pipe = pipe.fit(X_train, y_train)
    return pipe, pipe.score(X_test, y_test)
```

For the GitHub Gist, click [here](#).

Now we have a model, and we're ready to put together the app. All these functions will be run when the app runs because it's so fast that it doesn't make sense to save out a model object to be loaded. If your model doesn't train this fast, save your model object and return it in your app when you need to predict.

## Visualization

In addition to building a model and creating predictions, we want our app to show a visual of the prediction against a relevant distribution. The same plot function can be used for both apps, because we are using Plotly for the job.

The function below accepts the type of degree and the major, to generate the distributions, as well as the prediction that the model has given. That way, the viewer can see how their prediction compares to others. Later, we'll see how the different app frameworks use the Plotly object.

```
def plotly_hist(df, prediction=1, degreetype='All', majorfield = "All Fields"):  
    plot_series1 = df[df.CREDDDESC == degreetype]['EARN_MDN_HI_2YR'].astype(int)  
    plot_series2 = df[(df.CIPDESC_new == majorfield) & (df.CREDDDESC == degreetype)]  
    ['EARN_MDN_HI_2YR'].astype(int)  
  
    fig = go.Figure()  
    fig.add_trace(go.Histogram(x=plot_series1, name = "All Fields", histnorm='percent', xbins=dict(size  
= 1000)))  
    fig.add_trace(go.Histogram(x=plot_series2, name = majorfield, histnorm='percent', xbins=dict(size =  
1000)))  
  
    fig.update_layout(  
        bargroupgap=0.1,  
        title_text=f'Median Earnings, {degreetype}', # title of plot  
        xaxis_title_text='USD ($)', # xaxis label  
        yaxis_title_text='Percent of Total', # yaxis label  
        bargap=0.2, # gap between bars of adjacent location coordinates  
        bargroupgap=0.1, # gap between bars of the same location coordinates  
        template = "plotly_white",  
        font=dict(  
            family="Helvetica",  
            size=12),  
        legend=dict(yanchor = 'top', y=1,  
                    xanchor = 'right', x = 1)  
    )  
    fig.add_vline(x=prediction, line_dash="dash", annotation_text=f"Predicted: ${round(prediction,  
2):.2}")  
    fig.update_traces(opacity=0.55)  
    return fig
```

For the GitHub Gist, click [here](#).



This is the general visual we'll be generating – but because it's Plotly, it'll be interactive!



You might be wondering whether your other favorite visualization library could work here – the answer is, maybe. Every Python viz library has idiosyncrasies and is not likely to be supported exactly the same for Voila and Flask. I chose Plotly because it has interactivity and is fully functional in both frameworks, but you are welcome to try your own visualization tool and see how it goes.

## Deployment

Now, you're ready to move on to deploying the model. In Part 2, you'll learn to do it with Voila, and in Part 3, you'll learn the process with Flask.

## Part 2

# Voilà Web Application

In this part, we'll learn to build a fully functional web application just using a Jupyter notebook through the functionality of Voilà. Developed by the Project Jupyter community, this library allows users to share results and application functionality without the frontend expertise many other tools require.

The usefulness of Voilà is certainly not limited to machine learning models, but it happens to be a very convenient tool for deploying models without a steep learning curve.

When might you want to use this kind of application? One example might be when your audience needs to use your model in a self-service manner but doesn't have detailed technical expertise. By setting up a Voilà web application, you can just give your users the URL and they can be self-sufficient. This frees up data scientists' time for other work and ensures the business stakeholders get the insights they need fast.

Since you created the model in Part 1 of this series, you have everything you need to produce a deployment. Read on to learn how.

## Our Toolkit

[Saturn Cloud](#)

[Voilà](#)

[Plotly](#) (python and JS)

[Scikit-learn](#) (for our model)

## Other Helpful Links

[ipywidgets](#) (helpful for Voila)

[codebook for the dataset](#)

[plotly.js cheat sheet](#)

## See it in Action!

If you have a [Saturn Cloud account](#), you can [see this application running live now](#).

## Voila

Now, we are ready to start building our Voila app. Voila is entirely contained in a Jupyter notebook, so you'll need to start a new notebook, and add in all the functions we wrote up above. You don't need any other file types or frameworks, you can do it all in one Jupyter notebook. You can go ahead and add a chunk with calls for the modeling functions, so we are ready with our data.

```
df = load_data()
X, y = split_data(df)
modobj, modscore = trainmodel(X, y)
```

With this, we have our raw data, our model object, and the R-squared value.

Next, we start building the widgets – this is how your viewer will make selections to populate the prediction. In one or more chunks, add a widget for every input the user might want to make – here's one example. These can reference the items you've created already in the notebook – as we did here, that might mean using data from the dataset to populate a selector. (Sort this the way you want it to appear, because the widget is not going to reorder the list.)

```
creds = sorted(list(df['CREDESC'].unique()))

credential_widget = widgets.Dropdown(
    options=creds,
    value="Associate's Degree",
    description='Credential:',
    style = {'description_width': 'initial'},
    layout=widgets.Layout(width='80%')
)
```

For the GitHub Gist, click [here](#).

When you have all your widgets created and the app runs, your input selectors will look something like this.

Enter your choices...

Locale:
City: Large (population of 250,000 or more)

Region:
Far West (AK, CA, HI, NV, OR, WA)

Major:
Accounting and Related Services.

Credential:
Associate's Degree

College Type:
Public

Admitted Avg SAT Score:
800

Cost of attendance, USD:
15000

Admission Rate:
0.30

## Create Rendering Functions

The next task is to put all this together with functions that listen for user inputs, and complete a task – for us, this is predicting the earning value.

You can write one or many of these functions, but the key is how you call them in the `interactive_output` wrapper.

This function listens for all the input widgets, and when any of them change, it runs. It accepts all the user inputs, and forms them into a dataframe that is shaped correctly for the model object. It predicts on that dataframe, and then creates an HTML chunk using f-strings. Then it calls our plotting function, described earlier. It displays the HTML and the plot together, and this is what we'll see on the app.

```
def results2(sat, cred, cip, col, reg, tuit, loc, adm):
    newdf = pd.DataFrame([[sat, cred, cip, col, reg, tuit, loc, adm]],
        columns = ['SAT_AVG_ALL', 'CREDDDESC', 'CIPDESC_new', 'CONTROL',
            'REGION', 'tuition', 'LOCALE', 'ADM_RATE_ALL'])

    [[prediction]] = modobj.predict(newdf)
    pred_final = prediction if prediction > 0 else np.nan

    pred_html = f"""
    <hr>
    <h2> Model Predicts... </h2>
    Two years after graduating, median earnings should be
    roughly <b> ${round(pred_final, 2):.} </b>per year.

    """

    display(HTML(pred_html))
    p3 = plotly_hist(df=major, degreetype=cred, prediction=pred_final,
        majorfield=cip)
    display(go.FigureWidget(p3))
```

For the GitHub Gist, click [here](#).

However, recall that I mentioned we need to call it an `interactive_output`. If we don't, Voila doesn't know we mean for this to listen for inputs and run. But this is a simple task, just one call, as shown here. It calls the function and also passes a dict containing the arguments as keys and the widget names as values.

```
out2 = widgets.interactive_output(results2,
                                   {'loc': locale_widget,
                                    'reg': region_widget,
                                    'cip': cippick_widget,
                                    'cred': credential_widget,
                                    'col': coltype_widget,
                                    'sat': sat_widget,
                                    'tuit': tuit_widget,
                                    'adm': adm_widget})
```

For the GitHub Gist, click [here](#).

At this point, we're ready to do our page layout. You can certainly add lots of other design elements if you want to, and pass them as HTML or as other widgets.

## Page Layout

To do our layout, I'm using the Grid elements of Voila – but there are many ways to arrange a Voila app that are not shown here. As you might guess, VBox means vertical box and HBox means horizontal box, and those are the containers I'm using.

My vertical box is called `vb`, and it holds all my widgets for the user to interact with. My horizontal box, which will lay below this, is called `hb`, and it contains the `out2` interactive widget created earlier. “children” here just means the items that go inside each box. You can manipulate some layout elements for these boxes right here.

```
vb = widgets.VBox()
vb.children = [locale_widget,
               region_widget,
               cippick_widget,
               credential_widget,
               coltype_widget,
               sat_widget,
               tuit_widget,
               adm_widget]

hb = widgets.HBox(layout=Layout(width='auto', grid_area='hb'))
hb.children = [out2]
```

For the GitHub Gist, click [here](#).



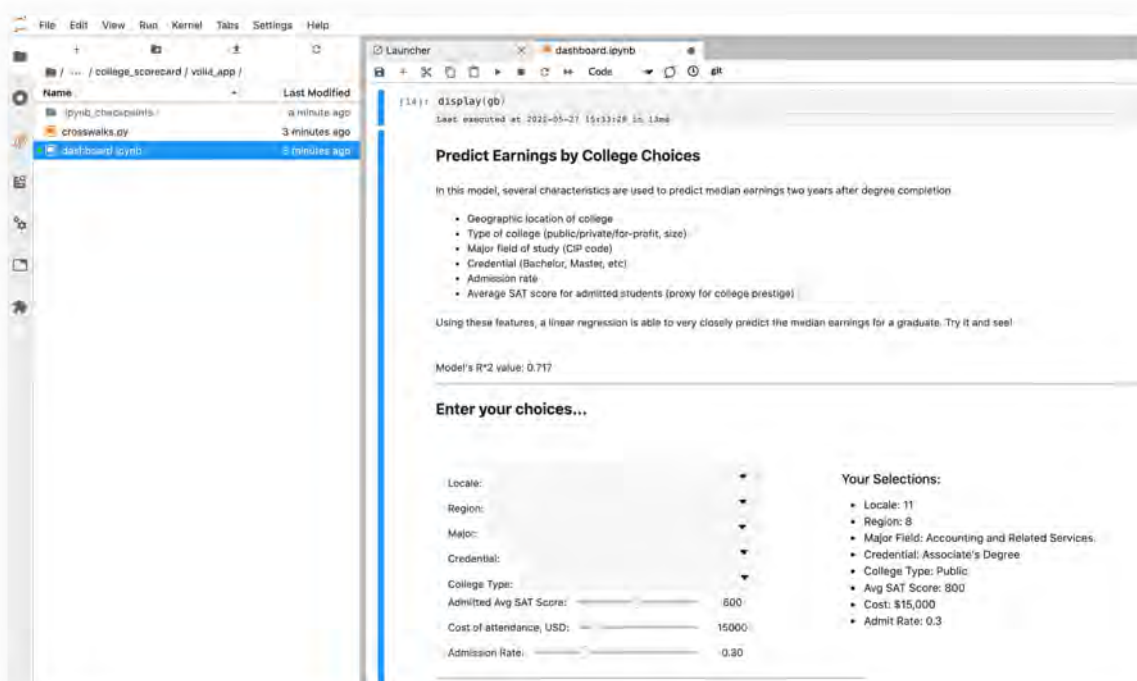
The next thing I want to do is create my actual Grid. I'm making `gb` my grid, and it contains the two objects `vb` and `hb`. The Grid spans 80% of the page width, and is 2x2 shaped. The columns are each half the size of the grid wide, and the grid template is described. Grid styling can be hard, but [there's a great tutorial with diagrams](#) that can help.

```
gb = widgets.GridBox()
gb.children = [vb, hb]
gb.layout.width='80%'
gb.layout.grid_template_rows='auto auto'
gb.layout.grid_template_columns='50% 50%'
gb.layout.grid_template_areas='''vb .
                                "hb hb"'''
```

For the GitHub Gist, click [here](#).

Now you're ready to run your app. Run this chunk, and your app should appear right in your notebook page!

```
display(gb)
```



That's nice, but it's not quite what we want our final product to be. To deploy this notebook for other users to see is a simple task with Saturn Cloud.

## Deploy

Make sure your code is in a Github repo you can access. Go to the Saturn Cloud project of your choice, and connect that repo to your [project](#). This is how your deployment will find the code.

Then return to the Saturn Cloud UI, and [open a project](#). Inside the project page, go [down to Deployments](#). You have the opportunity to customize a lot here, but the most important piece is the command – this needs to use the host 0.0.0.0 and the port 8000 to run on Saturn Cloud correctly. It's the same command you might use if running Voila locally on your laptop, just make sure the port and host are right. The path to your code is likely to start `../git-repos/` and will then be followed with the path to your file inside the repo you attached.

**Edit Deployment**

**Project**  
collegescorecard

**Name**  
collegescorecard

**Command**  
Choose a deployment type to pre-populate the command area:  
☐ Model ☐ Panel Dashboard ☐ Bokeh Dashboard ☐ Voila Dashboard  
voila --Voila.ip=0.0.0.0 --port=8000 ../git-repos/college\_scorecard/voila\_app/dashboard.ipynb  
This command will run from the working directory: /home/jupyter/project. Normally it's something like `python -m.py`

**Instance Count**  
1

**Hardware**  
☒ CPU ☐ T4 GPU ☐ V100 GPU

For this project, there's not much else you need to change, except that you should ensure that inside the Extra Packages section you note that plotly is required.

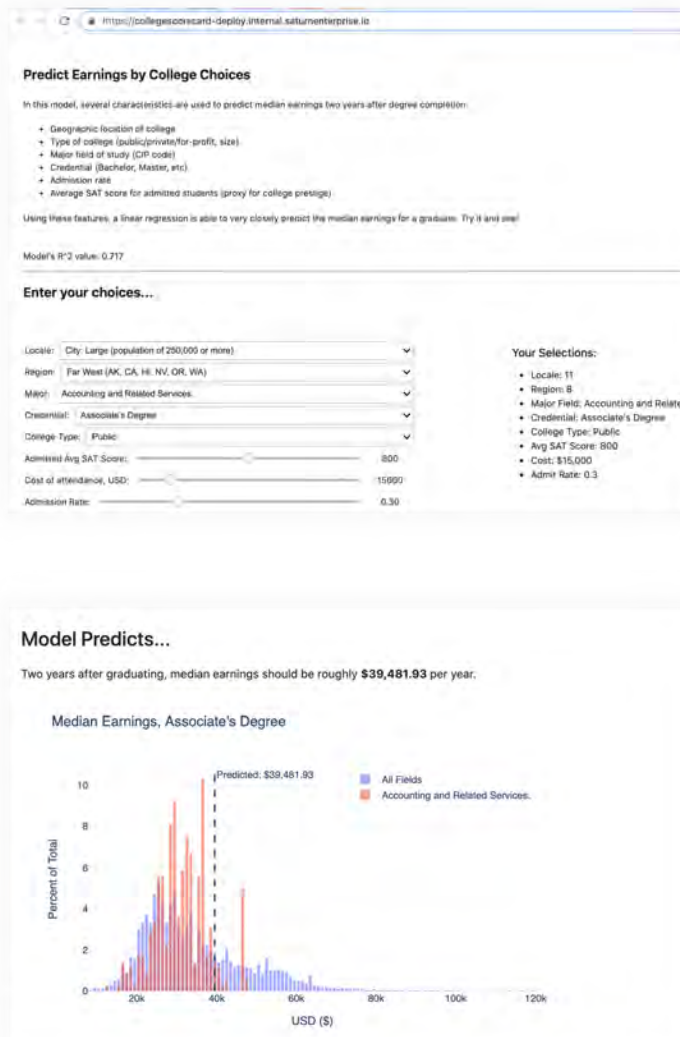
**Extra Packages**

Conda Install **Pip Install** Apt Packages

plotly

Extra packages will be installed when the project starts up - right before the `start_server`. Use spaces to separate multiple packages. Together they'll run the following script:  
`pip install plotly`

Now, save your deployment and hit the green Start arrow. It'll take a moment to start, and then the URL of that deployment should display your interactive app. Mine has a few extra text elements to explain the app, and you can write this sort of thing in yours too.



See it in Action!

If you have a [Saturn Cloud account](#), you can [see this application running live now](#).

## Conclusion

Congratulations - you deployed a model! This technique can work for lots of different types of models, and if you'd like to try it on Saturn Cloud, you can do so [here](#).

In Part 3, you'll learn the process of deploying your model as either a web application or a REST API with Flask.

## Part 3

# Flask API or Web Application

Flask is a Python-based web development framework with tremendously powerful tools. It may be a little more complicated to learn and use for the average data scientist than Voilà, but it offers more substantial flexibility in design and functionality.

In this next part, we'll see how you can employ Flask to serve a REST API or a fully developed web application, depending on your needs. Both start with the same steps, but you get to choose the presentation of the output that is right for you - either easily parsable JSON, or a full webpage.

The use case for a REST API will be different from a web application, in most cases. If your audience is business users, for whom raw JSON output might be off-putting or confusing, an API is not the right choice. However, in many cases your model's output will be fed to a database, another application, or combined with other information before its final use. In these cases, your results should be programmatically interpretable, and JSON is an excellent format for that. So, when you need your output to be parsed by computers, rather than being read by human users, choose an API!

## Our Toolkit

[Saturn Cloud](#) (so you can easily deploy)  
[Flask](#)  
[Plotly](#) (python and JS)  
[Scikit-learn](#) (for our model)

## Helpful Links

[codebook for the dataset](#)  
[Jinja](#) (helpful for Flask)  
[plotly.js cheat sheet](#)

### See it in Action!

If you have a [Saturn Cloud account](#), you can [see the Flask web application](#) running live now, or [see the Flask REST API](#).

Since you created the model in Part 1 of this series, you have everything you need to produce a deployment. Read on to learn how.

## Flask

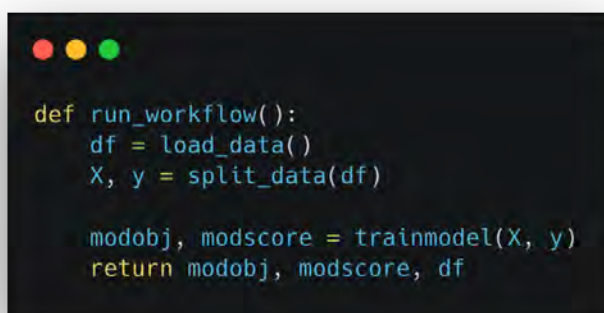
To build a deployment in Flask, we'll be using Python scripts instead of the Jupyter notebooks that Voila uses. As a result, there's more flexibility and possibility available – especially if you start integrating additional frontend frameworks. For this project, we have two choices about how to proceed:

- A bare REST API
- A web app (displaying an interface like we made with Voila in Part 2)

If we choose the web app, we'll be developing with Python, HTML, CSS, and some Javascript. That might sound like a lot, but it's less complex than it might seem at first. In the case of the API, you can avoid a lot of the front-end design, but some is still advisable to help your user get the hang of things.

## Application

As with Voila, we want to take the functions we wrote and put them into a script – instead of `.ipynb`, this will be just `.py`. The majority of our work can be contained in this Python script, which we'll call `app.py`. This holds all our code described above and in Part 1, as well as some functions depending on what kind of endpoint we're building. I'm going to add one extra function as well, which wraps up all the work we did and runs it together when called.



```
def run_workflow():  
    df = load_data()  
    X, y = split_data(df)  
  
    modobj, modscore = trainmodel(X, y)  
    return modobj, modscore, df
```

For the GitHub Gist, click [here](#).

Now, inside our script, we initialize Flask, call our model function, and get everything set up.

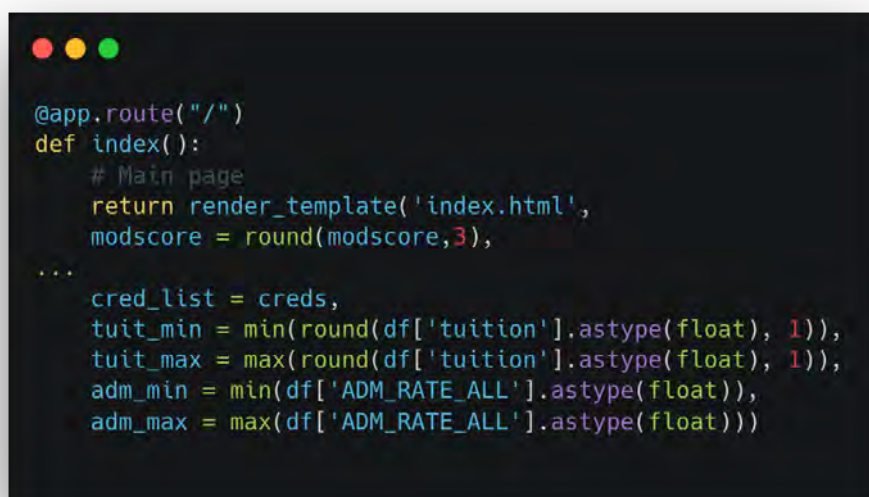
```
app = Flask(__name__)  
modobj, modscore, df = train_model()
```



## Index

It's probably nice to create a landing page for your users, whether they will be using your application in browser or not, just so there's somewhere to get information and learn to use your tool. In the app.py script, we'll have a function that creates this landing page.

In this function, called `index`, our Flask application is instructed to render the HTML- that's pretty much all this function does. We also have a few pieces of data in there to help the renderer make our value selectors correctly. (Abbreviated list for space.) Notice this function has NO returns- that's because its whole purpose is to make the HTML, and that's all.



```
@app.route("/")
def index():
    # Main page
    return render_template('index.html',
                           modscore = round(modscore,3),
                           ...
                           cred_list = creds,
                           tuit_min = min(round(df['tuition'].astype(float), 1)),
                           tuit_max = max(round(df['tuition'].astype(float), 1)),
                           adm_min = min(df['ADM_RATE_ALL'].astype(float)),
                           adm_max = max(df['ADM_RATE_ALL'].astype(float)))
```

For the GitHub Gist, click [here](#).

So it's generating some HTML – but how does it know what to do? We'll make a design template, and then fill in just the content we need to show the user.

What does the HTML template need to look like? It can start with the most basic HTML page outline. Flask uses Jinja templating to populate variables and pass information around, so you'll see a lot of items enclosed in double curly braces indicating Jinja variables. Also, note the section inside our body tags, because this is where any pages we make will populate the information they contain.

```
<!DOCTYPE html>

<html>
  <head>
    <meta charset="utf-8" />
    <title>Demo Model Deployment</title>
    <meta name="viewport" content="width=device-width, initial-scale=1" />
    <link rel="stylesheet" type="text/css" href="{{ url_for('static',filename='main.css') }}" />
    <script src="https://cdn.plot.ly/plotly-latest.min.js"></script>
  </head>

  <body>
    {% block content %}
    <!-- This is where our dynamic content will appear -->
    {% endblock %}
  </body>

  <footer>
    <h6>Courtesy of Saturn Cloud</h6>
    <script src="{{ url_for('static',filename='main.js') }}"></script>
  </footer>
</html>
```

For the GitHub Gist, click [here](#).

The index HTML page, as a result of the template, doesn't need to repeat the HTML wrappings at all. It can be as simple as some HTML description and the fields we are asking the user to select. It does, however, need to be wrapped in the following, so the templating system knows what to do.

```
{% extends "design.html" %} {% block content %}
<!-- our index-specific HTML and form -->
{% endblock %}
```

## Interactivity

If you want to make it possible for the user to submit options via the UI, then add a form and selectors to this page – if you just want people to submit options via URL arguments, you can skip this.

Almost everything you do in the index page then can be standard HTML. My selectors are enclosed in a form, and look like this. Notice that the credentials selector has several Jinja variables- how did these get here? From `app.py` !

```

<form action="/result">
  <div class="panel">
    <ul>
      <li>Credential:
        <select name='cred' method="GET" action="/">
          {% for credtype in cred_list %}
            <option value= "{{credtype}}" SELECTED>{{credtype}}</option>
          {% endfor %}
        </select>
      </li>
    </ul>
  </div>
  <div style="margin-bottom: 2rem;">
    <input type="submit" value="Submit" class="button"/>
    <input type="reset" value="Reset Values" class="button" />
  </div>
</form>

```

For the GitHub Gist, click [here](#).

Add as many selectors as your work requires.

## Results

Our next task is giving results back to the user. We can either return plain JSON as a REST API (easier to pass to other applications), or a UI version (easier to consume business users, for example). Whichever way we go, we need a function in app.py that will create the results from the inputs, however. This stub will do that for us. It loops over all our inputs, checks that they are present in the arguments on the URL, and then runs our prediction from the model.

```

@app.route('/result')
def result():
    variables = {'SAT_AVG_ALL': 'sat',
                'CREDESC': 'cred',
                'CIPDESC_new': 'cip',
                'CONTROL': 'coltype',
                'REGION': 'region',
                'tuition': 'tuit',
                'LOCALE': 'locale',
                'ADM_RATE_ALL': 'adm',
                }
    for i in variables:
        if variables[i] in request.args:
            if variables[i] == 'adm':
                variables[i] = request.args.get(variables[i], '', type= float)
            elif (variables[i] == 'tuit') | (variables[i] == 'sat'):
                variables[i] = request.args.get(variables[i], '', type= int)
            else:
                variables[i] = request.args.get(variables[i], '')
        else:
            return f"Error: No {variables[i]} field provided. Please specify {variables[i]}."

    newdf = pd.DataFrame([variables])

    [[prediction]] = modobj.predict(newdf)
    pred_final = prediction if prediction > 0 else np.nan
    ...

```

For the GitHub Gist, click [here](#).

## REST API

If we're not making a UI endpoint, then we can add this to our result function to complete the script.

```
...  
variables['predicted_earn'] = pred_final  
return jsonify(variables)
```

And that's that. When users visit our webpage, they'll get to submit options, or they can go directly to our URL and pass things programmatically. A JSON result will be returned. (You can customize the `@app.route` call to name it "api" or something similar that works for you.)

## UI Endpoint

If we need to present our results more attractively, we can make the output into a `result.html` page. The template we already saw for `index.html` can work for this too, so the HTML required is minimal. For fun, I'll also show you the plot rendering so we can replicate the Voila functionality fully.

First, this is the rest of the results function for `app.py` if we want to make a UI endpoint. It's not much more- we just create our plot (same function as Voila), and run `render_template` as we did for `index.html` with some different arguments.

```
...  
p3 = plotly_hist(df=df, degreetype = variables['CREDDESC'],  
                 prediction=pred_final, majorfield=variables['CIPDESC_new'])  
graphJSON = json.dumps(p3, cls=plotly.utils.PlotlyJSONEncoder)  
  
return render_template(  
    'result.html',  
    pred_final = round(pred_final, 2),  
    graphJSON=graphJSON  
)
```

For the GitHub Gist, click [here](#).

Our `results.html` code will receive all this stuff and display it for us. We can also pass all the model inputs if we want, and show them so the user sees all the values they provided - this is just optional.

For the rendering, then, all the results page really needs to have as far as HTML is this. It uses `pred_final` and `graphJSON` to print the prediction and also print the plot. We're using a little Javascript to print the plot, of course, but if you'd rather not get into that you can easily leave it out.

```
{% extends "design.html" %} {% block content %}
<h2> Model Predicts... </h2>
Two years after graduating, median earnings should be roughly
<b> {{ "${:,.2f}".format(pred_final)}} </b>per year.

<p style="font-size: 12px;"> If combination of values is
extremely rare/unlikely in training data, model may return NA.</p>

<div id="tester" style="width:900px;height:450px;"></div>

<script>
    TESTER = document.getElementById('tester');
    var GRAPH = {{ graphJSON | safe }}

    Plotly.plot(TESTER, GRAPH, {} );
</script>
{% endblock %}
```

For the GitHub Gist, click [here](#).

## Deploy

Now we have two applications: API and web app. Deploying them is actually just the same! Inside our `app.py` script/s we'll add one last bit (don't forget the port and host!):

```
if __name__ == "__main__":
    app.run(host='0.0.0.0', port=8000)
```

This lets us use the command `python app.py` to run our application.

Advanced Settings (optional) ^

Environment Variables

name = value

These are for your code only. There are no Saturn-specific environment variables you need to set. Environment variables should be of the form `VARIABLE=value`, and on a new line.

Start Script (Bash)

```
pip install Flask
pip install plotly
```

Any code you put in this box will be run every time a server is created from this project. If provided, the script will be run with bash. If you need to write to the file system, use `!cat` to view files.

Working Directory

/home/jovyan/git-repos/college\_scorecard/flaskapp

Save Cancel

Now you are ready! Save, click the green Start arrow, and your deployment will start up.

<https://college-score-flask-deploy.internal.saturnenterprise.io>

### Predict Earnings by College Choices

In this model, several characteristics are used to predict median earnings two years after degree completion.

- Geographic location of college
- Type of college (public/private/for-profit, size)
- Major field of study (CIP code)
- Credential conferred (Bachelor, Master, etc)
- College's admission rate
- Average SAT score for admitted students

Using these features, a linear regression is able to very closely predict the median earnings two years post graduation. Try it and see!

#### Enter your choices...

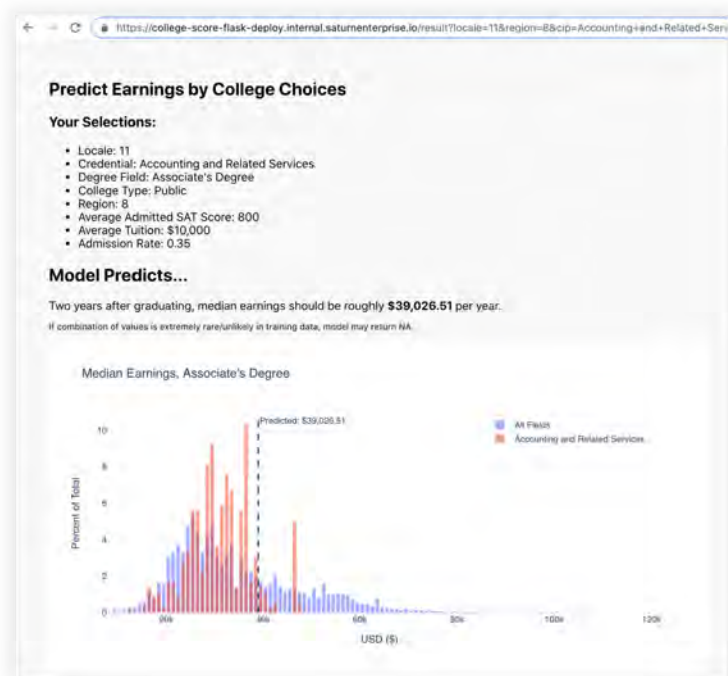
- Locale: City: Large (population of 250,000 or more)
- Region: Far West (AK, CA, HI, NV, OR, WA)
- Degree Field: Accounting and Related Services
- Credential: Associate's Degree
- College Type: Public
- Average Admitted SAT Score: 800
- Average Tuition: 10000
- Admission Rate (Percent): 35

Submit Reset Values

Model R<sup>2</sup>: 0.717

This form will accept your viewer's inputs, and then it will either return JSON results, if you're making an API, or will take them to your results page.





## See it in Action!

If you have a [Saturn Cloud account](#), you can the [Flask web application](#) running live now or see the [Flask REST API](#).

## Conclusion

With that, you have deployed a model. This technique can work for lots of different types of models, and if you'd like to try it on Saturn Cloud, you can do so [here](#). Check out our [getting started documentation](#) for more guides, too.

With that, you have deployed a model! This technique can work for many different types of models, as well as other data science and machine learning insights you might want to share with others.

Delivering value through productionizing models is an incredibly powerful aspect of machine learning in business and other contexts, and it's easier than ever to do it, thanks to tools like the ones you've learned here.

Saturn Cloud is eager to help you turn your models into insights. To do that, we offer one-click deployments, as well as access to as much or as little, compute as you require, simple collaboration tools, and familiar, easy development environments. The deployments we've built in this guide are running on Saturn Cloud right now, and you can view them if you have a Saturn Cloud Hosted account!

- [Voilà webapp](#)
- [Flask web app](#)
- [Flask API](#)

## About Saturn Cloud

Saturn Cloud is a free, end-to-end data science and machine learning platform allowing data scientists to scale their Python projects with Dask in the cloud and more.

If Saturn Cloud might be right for your needs, [check out our documentation and learn how to get started in 10 minutes or less, with clear and accessible guides and how-tos](#). We look forward to seeing what exciting projects you build on Saturn Cloud!



# Your Data Science Best Friend



## ALL-IN-ONE WORKSPACE

- Scalable Python with GPUs & Dask
- Cloud-hosted Jupyter Notebooks
- Sharable work & dashboards
- Familiar, easy development environments
- One-click deployments
- Access to as much compute needed
- Simple collaboration tools
- Connect from existing cloud-hosted services

### Trusted by



And More

Join thousands of data scientists  
creating faster and more scalable  
Python projects and more on  
Saturn Cloud

START FOR FREE