How to Create and Sell Your Machine Learning Product Online and For Free

Don't wait for magical inspiration or a partner in white shining armor to invite you - instead, start entrepreneuring today and for free!

(Source: Lucas Amunategui)

Table of Contents

Introduction

You read the title correctly, for free. That's the best price possible!

Mind you, this isn't such a big stretch as the Internet, open-source tools, online guides, etc. have either been getting cheaper by the day or have already hit zero, i.e. free. This is the great democratization of data and computer science. Let me show you how to build one of these free pipelines for your machine-learning projects.

What isn't going to lose value is your intellectual property, that is, only if you are able to share them to the world. You don't want to waste or limit the extent of all your brilliant ideas you have created using your machine learning, artificial intelligence, and data crunching pipelines. In this eBook, I'll show you how to easily extend those ideas from a standalone python script, into a fully automated pipeline that doesn't only end in an awesome, fully interactive web application, but doesn't end at all. I'll go over simple metrics (Google Analytics), to get the feedback needed to iterate new versions and reach back to your users and keep extending that relationship and making your ML product better.

For those that are familiar with my material (amunategui.github.io and ViralML.com), this will be a concise version of a lot of the things I've been talking about. Everything starts at the drawing board. We'll start by exploring an idea, developing it, making sure it has value, and script a standalone version. In most cases you will already have that as that his your standalone python script. Once you have determined that each function runs properly and returns the expected values, we will extended it into a local flask version. This will be very similar to the final web product on the cloud except that it only runs on a local machine.

Once we're happy with the local Flask application and can confirm that everything is working properly, we're ready for the cloud! There will be unique

customization points to factor in according to the cloud provider (especially with free services that come with inherent limitations). This is normal and nothing to worry about. As with any pipeline with lots of moving parts, there will be things to fix along the way.

And the story doesn't end there. Once you're application is up and running, you will need to drive traffic to it, analyze usage, promote, improve, up-sell, etc. We'll briefly cover some of those topics like blogging, search optimization work (SEO), promotions and value-added content, interacting with users, adding features, mail campaigns, etc.

Phew, and all this for free, even this guide is free!

Are You Ready?

So, you're ready to give birth to your digital product? Are you sure? This is an important step, few come back from this. Once you get a taste, once you get an understanding of how easy it is to get something out there you will be hooked. You will question others around you who have either failed or have needed an armada of employees to get things going.

For simple, clear, intuitive ideas, the bridge from concept to reality is just a brisk and easy walk. And in our case, free! Yes, if you are willing to forgo some of the bells and whistles, you can launch, experiment and measure a site at no cost. And that is exactly what we will focus on here.

We're going to create a financial portal that will give stock advice based on data science and machine learning (BTW, this is only a toy project, don't go trading on this stuff). We will leverage free sources of data, apply machine learning in order to find patterns and alert our readers. As this is an entirely free web application, we'll use a stripe - pay-as-you-go option plan and/or a PayPal donate to solicit

support.

So, don’t wait for magical inspiration or a partner in white shining armor to invite you - instead, start entrepreneuring today and for free!!!

Wait, One More Thing

Obviously, when I say for free, I mean no cash required, but, as we all agree with the adage that 'time is money', then there is still a cost. Me, writing this book costs me money. You reading this book costs you money. So figure out your hourly wage and make sure you’re comfortable with that cost. Thankfully, for us weekend entrepreneurs, we are more than willing to forgo nights out and weekends at the beach to make our dreams of releasing a digital product a reality. So there is a cost, just not a monetary one - keep that in mind and make sure this is how you want to spend your precious time!

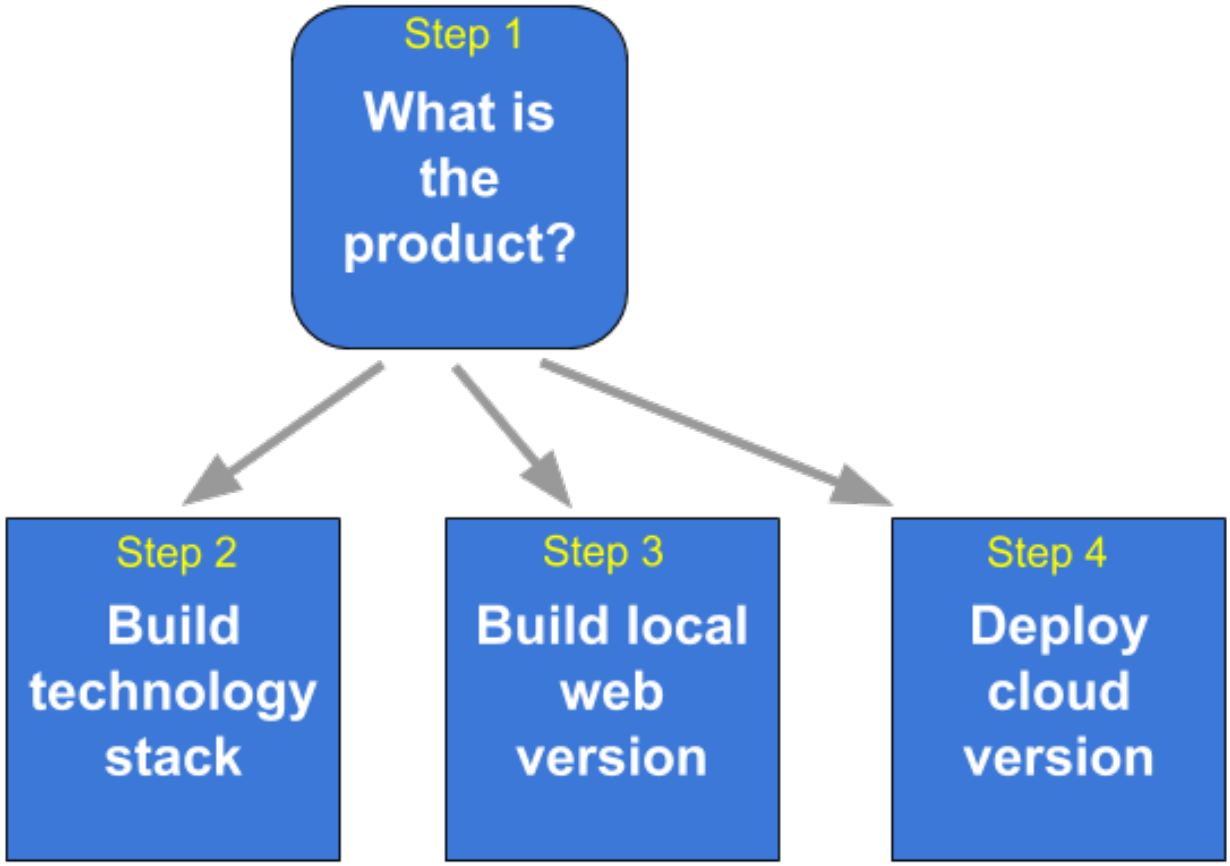
The Source Code (and Video Links)

As you navigate through each section, you will see links to the source code. As I’ll only be showing short snippets in this eBook, you’ll need to copy and paste the code for each section to your local machine to follow along.

The Process

We start at the end, by understanding what the user wants to see. Step 1 is where we invest time and thought on the final goal and user experience. We ensure that the appropriate modeling approach is used in order to reach that web-

application state quickly and without surprises.



Next we build our python scripts, design algorithms, abstract functions and test outputs, this is basically your web application without the web, this is Step 2.

In Step 3, we design and develop the local web application. This step requires leveraging various web front-end technologies to offer the needed interactivity and dynamism to highlight a model's insight and sustain a user's interest. The final product at this stage is indistinguishable from the next one except that it is hosted on your local machine, not the cloud.

Finally, we deploy onto a popular, reliable and free cloud provider. This is the final stage, Step 4, where the world gets to enjoy and learn from your work.

The Web Application Product

A digital product can be many things from a complex cloud application like Microsoft Office 365 to a simple informational site like a static WordPress page.

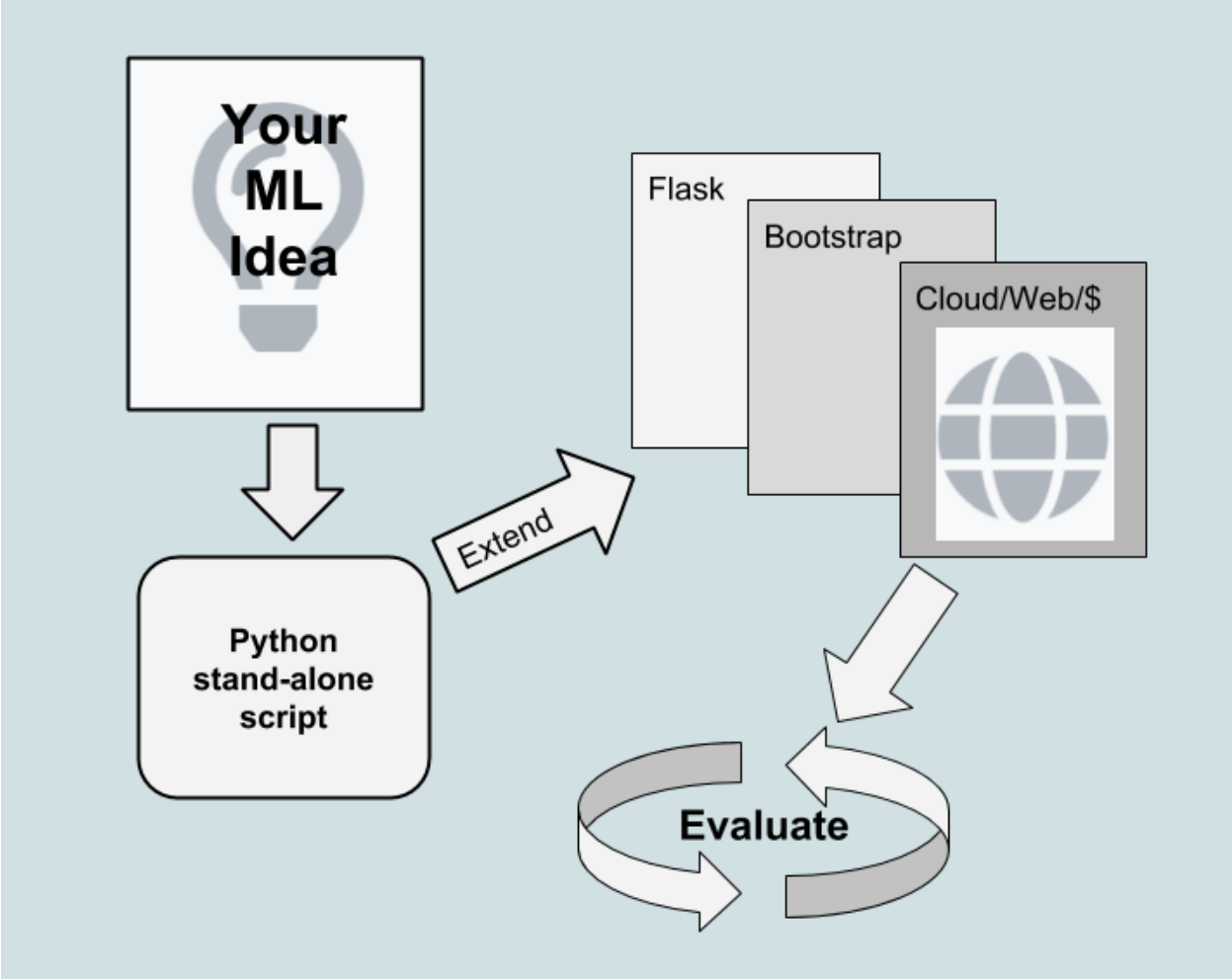
I will use one of my favorite style of digital products to illustrate these topics, the 'Machine Learning Web Application'. This is where you get to extend your Python data producing application into a full-fledged marketing, paywall or subscription selling site.

I have written a lot about these, even wrote a 450 page-book on Apress focusing entirely on that approach: 'Monetizing Machine Learning: Quickly Turn Python ML Ideas into Web Applications on the Serverless Cloud'.

I must say that I really like the digital product. It is a great way of getting a business going as it requires little investment besides time, has a crazy profit margin, can be duplicated at will, improved on the fly, controlled and supported from anywhere around the world. And if it is well designed and can pass the test of time, it becomes a passive income source, you know, that holy-grail, evergreen money-producing machine.

High-Level Overview of the Architecture

Our web application is the culmination of a series of events that form a digital pipeline resulting in a dynamic, interactive, and intelligent output. It will usually take some user's instructions, crunch some numbers, and present back a value-added answer.



The chart illustrates the process of showing how you start with an ML web application idea, build it out, deploy it to the Internet, and finally, evaluate its success.

The Sequential Development Process

Before moving anything to the cloud, we need to spend as much time as possible building, developing and debugging the code locally. We also need to abstract all the code into clean and logical functions. Having as much of the logic and mechanics figured out and coded before building any of the web trappings or cloud pieces, will save you huge amounts of time. The idea is that you want to

work out as many kinks and bugs within your local Python scripts and your local Flask web applications before thinking about the cloud. By knocking out all issues locally, will make your life a whole lot easier before uploading anything to the cloud.

To put it another way, if it doesn't work locally, it will work even less in the cloud.

Let's Get to Work

The Source Code (and video links)

As you navigate through each section, you will see links to the source code. As I'll only be showing short snippets in this eBook, you'll need to copy and paste the code for each section to your local machine to follow along.

I also have videos of this course for those that are more visual. It is free and can be found on my Teachable site:

https://ml-entrepreneur.teachable.com/purchase?product_id=831961

This is it, no more theory, let's roll up our sleeves and build a web application!

As with any product, from a web application to virtually anything else, we need to have a crystal clear idea of what we're going to build before starting.

Personally, I always start by thinking of what is the product and what is its purpose. I also make lists of immediate goals in the context of bigger ones, and take a periodic step back to make sure I can still tell the forest from the trees. It is critical to regularly calibrate your bearings.

If I can't do that, I stop and walk away from the computer. I think back at what the customer wants, what would constitute a good day for them, and how could this tool enable that. And it doesn't hurt to give your body a stretch and your eyes a break

Getting Started using a Clear Functional Direction

We can make this long story shorter, I've had a long talk with our fictional customer and they shared their desire to create a stock market prediction web application service that can forecast the price of a stock in the next three trading periods. Of course, they want this application to be accessible from anywhere so that any customer, no matter where they are located can use the service (yes, prefect for a web application!).

The product we're going to build is a bit besides the point here. We are really interested in developing a pipeline template that you can re-use for your own "real" needs. Nevertheless, it is still a very important piece as it will decide the type of model to use, the type of predictions to yield and will definitely dictate look and functionality of the UI.

Predicting The Stock Market

Getting Historical Data

(Link to code: https://manuelamunategui.pythonanywhere.com/eBook-code-ml-for-free-eda.html)

We're going to use a data set I've used many times in various projects of mine. Let's download some financial stock data from Kaggle's open data source. The dataset holds stock market end-of-day prices and corollary data for stocks of the S&P 500 over a period of 5 years. This is going to be perfect to train a model in predicting the stock market. Run the below snippet of code to download all the data we'll need to train this end-of-day (EOD) model on real stock market data. Notice the "Load locally" flag, turn that to "True" after running at least once. This will download the data set from Kaggle's GitHub repository once and save it locally so you can pull it directly from your hard-drive for subsequent runs (this will also make you a better what is netizen (net citizen) and allow your notebooks to run much faster).

```
local = False
if local:
     # load locally
     stocks_df = pd.read_csv('all_stocks_5yr.csv')
else:
     # Load directly from GitHub
     stock_data_file = 'https://raw.githubusercontent.com/CNuge/
kaggle-code/master/stock_data/all_stocks_5yr.csv'
    stocks_df = pd.read_csv(stock_data_file,
parse_dates=['date'])
    # make a local copy
    stocks_df.to_csv('all_stocks_5yr.csv', index=None)

stocks_df.head()
```

	date	open	high	low	close	volume	Name
0	2013-02-08	15.07	15.12	14.63	14.75	8407500	AAL
1	2013-02-11	14.89	15.01	14.26	14.46	8882000	AAL
2	2013-02-12	14.45	14.51	14.10	14.27	8126000	AAL
3	2013-02-13	14.30	14.94	14.25	14.66	10259500	AAL
4	2013-02-14	14.94	14.96	13.16	13.99	31879900	AAL

Let's perform some basic exploratory data analysis (EDA) to get better acquainted with the data.

Date Ranges

```
1 np.min(stocks_df['date'])
```

```
'2013-02-08'
```

```
1 np.max(stocks_df['date'])
```

```
'2018-02-07'
```

Size

```
1 stocks_df.shape
```

```
(619040, 7)
```

Description

```
1 stocks_df.describe()
```

	open	high	low	close	volume
count	619029.000000	619032.000000	619032.000000	619040.000000	6.190400e+05
mean	83.023334	83.778311	82.256096	83.043763	4.321823e+06
std	97.378769	98.207519	96.507421	97.389748	8.693610e+06
min	1.620000	1.690000	1.500000	1.590000	0.000000e+00
25%	40.220000	40.620000	39.830000	40.245000	1.070320e+06
50%	62.590000	63.150000	62.020000	62.620000	2.082094e+06
75%	94.370000	95.180000	93.540000	94.410000	4.284509e+06
max	2044.000000	2067.990000	2035.110000	2049.000000	6.182376e+08

Number of Unique Symbols

```
1 print('Number of unique symbols:', len(set(stocks_df['Name'])))
```

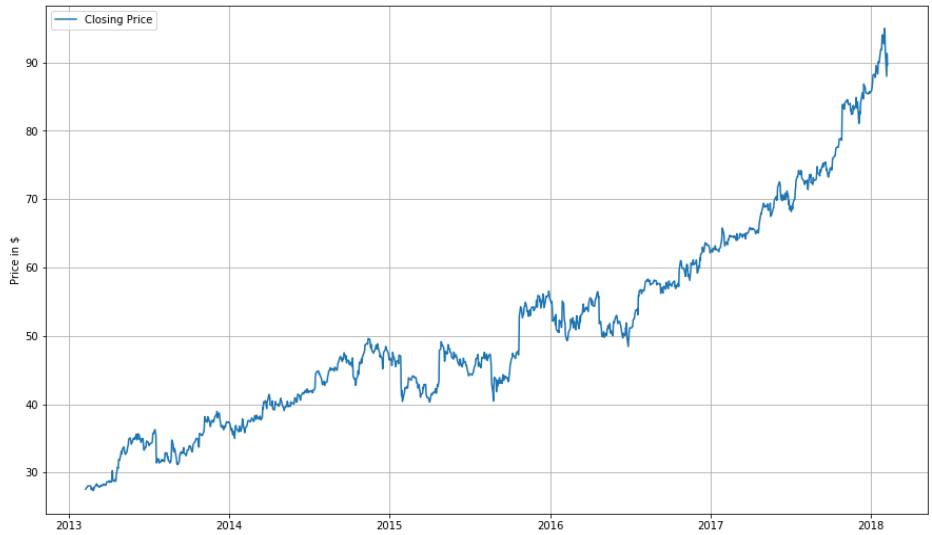
```
Number of unique symbols: 505
```

Spot Check

And for fun, let's plot one of these. Let's go with Microsoft Corporation.

```
fig, ax = plt.subplots(figsize=(15,9))
temp_df = stocks_df[stocks_df['Name']=='MSFT']

ax.plot(pd.to_datetime(temp_df['date']), temp_df['close'],
label='Closing Price')
ax.grid()
ax.legend(loc='best')
ax.set_ylabel('Price in $')
```



Getting Our Data Model-Ready

(Link to code: https://manuelamunategui.pythonanywhere.com/eBook-code-ml-for-free-prep-data.html)

Let's use 10 periods of trading market and predict 5 days out (and please experiment with those settings as that's half the fun). This is time-series data so we are going to take consecutive segments for each stock to use as features. This means we are going to take every closing price, tie the previous 10 closing prices for that stock and add them as features to the row. This will allow each row to capture some of the price movement over the past couple of trading periods.

The idea behind this approach is based on trend following. There is a higher probability that if today is an up-day, tomorrow will be an up day as well. So here we'll give the model 10 days to learn what is the overall direction of the market and predict accordingly.

We will also help out the model and normalize the prices by applying the log transform on all of them (we will have to translate it back when presenting data

to the customer).

Running this portion of the code takes a couple of minutes, so be patient as it has to perform this on 500 individual stocks.

```
stock_symbols = list(set(stocks_df['Name']))

# build dataset
X = []
y = []
symbols = []
prediction_dates = []
last_market_dates = []

# rolling predictions
rolling_period = 10
predict_out_period = 5

# loop through each stock to gather time-series data only
# for that type and not mix
for stock in stock_symbols:
    print(stock)
    stock_data = stocks_df[stocks_df['Name']==stock].copy()

    for per in range(rolling_period, len(stock_data)-
predict_out_period):
        X_tmp = []
        y_tmp = 0
        for rollper in range(per-rolling_period,per):
            # build the 'features'
            X_tmp +=
[np.log(stock_data['close'].values[rollper])]

        X.append(np.array(X_tmp))
        # build 'labels'
        y.append(np.log(stock_data['close'].values[per +
predict_out_period]))
        prediction_dates.append(stock_data['date'].values[per +
predict_out_period])
        last_market_dates.append(stock_data['date'].values[per])
        symbols.append(stock)

# package it all into a neat dataframe
stock_model_ready_df = pd.DataFrame(X)
stock_model_ready_df.columns = [str(f) for f in
list(stock_model_ready_df)]
```

```
stock_model_ready_df['outcome'] = y
stock_model_ready_df['prediction_date'] = prediction_dates
stock_model_ready_df['last_market_date'] = last_market_dates
stock_model_ready_df['symbol'] = symbols
```

Now you will notice that we have one row per market date (last_market_date) made up of a sequence of 10 periods of trading prices. It also has the 5 periods out as our outcome variable (i.e. what the model will attempt to predict) and the date of the prediction along with the symbol.

```
1 stock_model_ready_df.head()
```

	0	1	2	3	4	5	6	7	8	9	outcome	prediction_date	last_market_date	symbol
0	4.148675	4.158883	4.170843	4.161536	4.156850	4.156850	4.151197	4.145830	4.133405	4.141546	4.151355	2013-03-04	2013-02-25	PNC
1	4.158883	4.170843	4.161536	4.156850	4.156850	4.151197	4.145830	4.133405	4.141546	4.107919	4.153556	2013-03-05	2013-02-26	PNC
2	4.170843	4.161536	4.156850	4.156850	4.151197	4.145830	4.133405	4.141546	4.107919	4.113166	4.161380	2013-03-06	2013-02-27	PNC
3	4.161536	4.156850	4.156850	4.151197	4.145830	4.133405	4.141546	4.107919	4.113166	4.121960	4.171460	2013-03-07	2013-02-28	PNC
4	4.156850	4.156850	4.151197	4.145830	4.133405	4.141546	4.107919	4.113166	4.121960	4.133405	4.174695	2013-03-08	2013-03-01	PNC

Splitting the Data in to Train, Test, and Validation Portions

(Link to code: https://manuelamunategui.pythonanywhere.com/eBook-code-ml-for-free-model-and-evaluate.html)

As with most modeling projects, we need to split our data into different portions: a training, a testing and a validation set. By doing this, we can use one portion to teach the model how to predict the stock market, the other portion to evaluate the model, and we'll keep all the data for one single stock as a validation set and to plot the actual versus predicted - visuals are always good for your stakeholders.

Two Approaches to Splitting Data

Option 1: Random Splits

The "**train_test_split()**" function from "**sklearn**" is great as it does most of that for us. Setting the seed (random_state) is paramount for reproducibility as the function will shuffle the data randomly before splitting it. But by using the same seed you will always get the same split in subsequent runs.

Option 2: Split By Date

There isn't a one-size-fits-all, in some cases, it is good to split randomly, in others, where the time element can be intersting, then splitting it by time chunks can be useful. The code below does both, so please experiment a lot!

We also create a validation data set by taking out all data for a particular stock symbol and we'll use the model to predict ever day for that stock and plot the predictions with an overlay of the actual price. Visuals! A visual makes for a thousand words…

```
from sklearn.model_selection import train_test_split

TARGET = 'outcome'
FEATURES = [f for f in list(stock_model_ready_df) if f not in
[TARGET, 'prediction_date', 'last_market_date', 'symbol']]

# keep on stock for out of sample testing - you can change this
to your
# favorite stock
oos_stock ='MMM'
oos_stock_df =
stock_model_ready_df[stock_model_ready_df['symbol']==oos_stock]
print('live excluded test data:', oos_stock_df.shape)
stock_model_ready_df =
stock_model_ready_df[stock_model_ready_df['symbol']!=oos_stock]

# time based split or random?
TIME_BASED = True
if TIME_BASED:
    # date split
    date_split = '2017-01-01'
    x_train =
stock_model_ready_df[stock_model_ready_df['last_market_date'] <=
```

```
date_split]
    x_test =
stock_model_ready_df[stock_model_ready_df['last_market_date'] >
date_split]

    # shuffle training data
    x_train = x_train.sample(len(x_train))
    y_train = x_train[TARGET]
    y_test = x_test[TARGET]
else:
    x_train, x_test, y_train, y_test =
train_test_split(stock_model_ready_df,
                                    stock_model_ready_df[TARGET],
test_size=0.33, random_state=42)

print('x_train.shape:', x_train.shape)
print('x_test.shape:', x_test.shape)
```

```
live excluded test data: (1244, 14)
x_train.shape: (473916, 14)
x_test.shape: (136305, 14)
```

Let's Learn About the Stock Market Using XGBoost

XGBoost is a phenomenal model. It can do classification, regression, multi-class classification, and more. It is one of the top models on Kaggle.com as it keeps on winning competitions! It runs on all sorts of programming languages - Julia, R, Python… For plenty more details about this incredible model, see the official docs at: https://xgboost.readthedocs.io/en/latest/

Anyway, it's a great and fast model, perfect for our needs here.

```
import xgboost as xgb

dtrain = xgb.DMatrix(data = x_train[FEATURES],  label = y_train)
dval = xgb.DMatrix(data = x_test[FEATURES], label = y_test)

param = {'max_depth':3,
            'eta':0.05,
```

```
            'silent':0,
            "objective":"reg:linear",
            "eval_metric":"rmse",
            'subsample': 0.8,
            'maximize': False,
            'colsample_bytree': 0.8}

evals = [(dtrain,'train'),(dval,'eval')]
stock_model = xgb.train ( params = param,
              dtrain = dtrain,
              num_boost_round = 1000,
              verbose_eval=50,
              early_stopping_rounds = 500,
              evals=evals)
```

XGBoost requires the data be cast to its DMatrix format. You can also break out model parameters into a dictionary to keep things organized and readable. Check out the documentation for more details on the parameters - the important ones to tweak are "**max_depth**" and "**eta**".

When you run the model, it will print out a progress log and, as this is a linear model, you are looking for a decreasing root mean square error (RMSE) value from one line to the next. If it doesn't, the model will trigger its "**early_stopping_rounds**" and stop. We end up with an RMSE of 0.03721 with doesn't tell us much as it is a log transform value. And even if we took the exponent of the value, which is 1.03, which means the model is off by +/- $1.03, its hard to translate that to the wide range of dollar values of each stock. It may be better to run a model on percentage difference as that may make the unit of each stock a little more comparable.

Anyway, we can then run the handy "**get_fscore**" function that will order out the most important features first. It is interesting that the first one listed is that last value, i.e. the most current in the sequence.

```
importances = stock_model.get_fscore()

importance_frame = pd.DataFrame({'Importance':
list(importances.values()), 'Feature': list(importances.keys())})
importance_frame.sort_values(by = 'Importance', inplace = True,
```

```
ascending= False)
importance_frame
```

	Feature	Importance
0	9	1668

Evaluating the Model

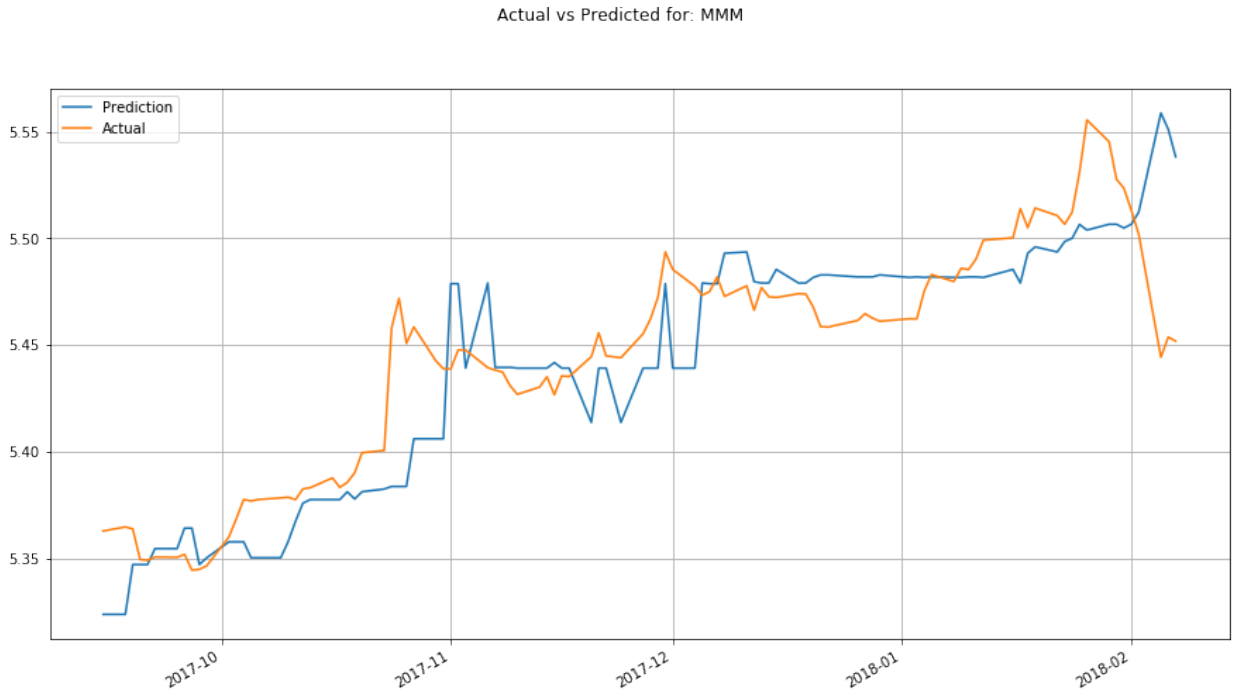
Books have been written on evaluating models. Here, we'll take a shortcut and simply apply the model on our validation data (that's the stock we earmarked purposely for visual evaluation) and plot the predictions against the actual values for comparison (which makes for a fantastic chart).

```
# evaluate on one stock only for clarity
tmp_df = oos_stock_df.tail(100)
doos = xgb.DMatrix(data = tmp_df[FEATURES],  label =
tmp_df[TARGET])

# predictions
preds = stock_model.predict(doos)

# get accuracy of score
fig, ax = plt.subplots(figsize=(15,8))
plt.plot(pd.to_datetime(tmp_df['prediction_date']), list(preds),
label='Prediction')
plt.plot(pd.to_datetime(tmp_df['prediction_date']),
list(tmp_df[TARGET]), label='Actual')
plt.legend()
plt.suptitle('Actual vs Predicted for: ' + oos_stock)
plt.grid()
plt.xticks(rotation='vertical')
ax.xaxis_date()
fig.autofmt_xdate()
```

```
plt.show()
```



Not too bad, right? Would I trade on these signals? Definitely not. Keep in mind that the scaling is still the log transform and in order to show this to web viewers, we will need to take the exponent to translate it back to normal dollar scale.

Also, again, this is only a toy project so don't trade this!

Good Enough, but How About Testing This on Live Data?

(Link to code: https://manuelamunategui.pythonanywhere.com/eBook-code-ml-for-free-live-data.html)

Yes, that's the next big question and we're going to get data from one of the online providers that still offers stock market data. Providers can be flaky (its a free service so you can't complain). The good thing is we are predicting a few days into the future, so even if you miss a day or two, you'll still be able to offer a prediction.

One way to do this is using a site that offers a data download URL like financialcontent.com that automatically returns a CSV file.

```
live_stock_data_raw = pd.read_csv('http://
markets.financialcontent.com/stocks/action/gethistoricaldata?
Month=10&Symbol=MSFT&Range=50&Year=2018')
```

The above line will return 50 trading periods of Microsoft Corporation from October 2018 backwards.

```
live_stock_data_raw.head()
```

	Symbol	Date	Open	High	Low	Close	Volume	Change	% Change
0	MSFT	10/03/18	115.42	116.18	114.93	115.17	16644772	0.02	0.02%
1	MSFT	10/02/18	115.30	115.84	114.44	115.15	20780726	-0.46	-0.40%
2	MSFT	10/01/18	114.75	115.68	114.73	115.61	18881814	1.24	1.08%
3	MSFT	09/28/18	114.19	114.57	113.68	114.37	21647800	-0.04	-0.03%
4	MSFT	09/27/18	114.78	114.91	114.20	114.41	19058123	0.43	0.38%

Unfortunately, PythonAnywhere won't allow us to do this can of data pulling on a free-tiered account (one of the limitations of going free). But fear not, we will use the trusted (but experimental) Pandas function "pandas_datareader.data" that will pull data from the web site https://robinhood.com/. Let's take a quick look at the data:

```
symbol='MSFT'
import pandas_datareader.data as web
```

```
start = datetime.datetime(2018, 1, 1)
end = datetime.datetime.now()
live_stock_data = web.DataReader(symbol, 'robinhood', start, end)
live_stock_data.reset_index(inplace=True)
live_stock_data.head()
```

```
  symbol  begins_at close_price high_price  interpolated  low_price  \
0   MSFT 2017-10-09   74.932700  75.188100         False  74.510400
1   MSFT 2017-10-10   74.932700  75.266700         False  74.785400
2   MSFT 2017-10-11   75.060400  75.099700         False  74.598800
3   MSFT 2017-10-12   75.747900  75.914900         False  75.011300
4   MSFT 2017-10-13   76.111400  76.484600         False  75.914900

  open_price session    volume
0  74.618400     reg  11386502
1  74.972000     reg  13944545
2  75.001500     reg  15388898
3  75.129100     reg  16876538
4  76.209600     reg  15335742
```

Let's determine what we need to do to get in the same format as the training data. Let's start by dropping and renaming some of the columns.

```
live_stock_data = live_stock_data[['symbol', 'begins_at',
'close_price']]
live_stock_data.columns = ['symbol', 'date', 'close']
 live_stock_data.head()
```

```
  symbol       date      close
0   MSFT 2017-10-09  74.932700
1   MSFT 2017-10-10  74.932700
2   MSFT 2017-10-11  75.060400
3   MSFT 2017-10-12  75.747900
4   MSFT 2017-10-13  76.111400
```

We also need to apply the log transform of the closing price, and add the time-series features.

```
# apply the log transform
live_stock_data['close'] = np.log(live_stock_data['close'])
# sort by ascending dates as we've done in training
live_stock_data = live_stock_data.sort_values('date')
live_stock_data.head()
```

	symbol	date	close
1028	MSFT	2014-09-03	3.805773
1027	MSFT	2014-09-04	3.812424
1026	MSFT	2014-09-05	3.826683
1025	MSFT	2014-09-08	3.838807
1024	MSFT	2014-09-09	3.845028

Now we need to create the same 10 period sequences per row as we did on the training data set. This time we don't have to worry about the outcome variable as we are going to predict the real future where prices aren't known yet (and if I did, I'd be super rich!!).

```
# build dataset
X = []
y = []

prediction_dates = []
last_market_dates = []

# rolling predictions
rolling_period = 10
predict_out_period = 5

for per in range(rolling_period, len(live_stock_data)):
    X_tmp = []
    y_tmp = 0
    for rollper in range(per-rolling_period,per):
        # build the 'features'
        X_tmp += [live_stock_data['close'].values[rollper]]

    X.append(np.array(X_tmp))

    # add x days to last market date using numpy timedelta64
    prediction_dates.append(live_stock_data['date'].values[per] +
np.timedelta64(predict_out_period,'D'))
    last_market_dates.append(live_stock_data['date'].values[per])
```

```
live_stock_ready_df = pd.DataFrame(X)
live_stock_ready_df.columns = [str(f) for f in
list(live_stock_ready_df)]

live_stock_ready_df['prediction_date'] = prediction_dates
live_stock_ready_df['last_market_date'] = last_market_dates
```

```
1 live_stock_ready_df.tail()
```

	0	1	2	3	4	5	6	7	8	9	prediction_date	last_market_date
1014	1.553204	1.554064	1.551755	1.553765	1.550922	1.554437	1.555716	1.556471	1.556066	1.555198	2018-10-02	2018-09-27
1015	1.554064	1.551755	1.553765	1.550922	1.554437	1.555716	1.556471	1.556066	1.555198	1.555993	2018-10-03	2018-09-28
1016	1.551755	1.553765	1.550922	1.554437	1.555716	1.556471	1.556066	1.555198	1.555993	1.555919	2018-10-06	2018-10-01
1017	1.553765	1.550922	1.554437	1.555716	1.556471	1.556066	1.555198	1.555993	1.555919	1.558191	2018-10-07	2018-10-02
1018	1.550922	1.554437	1.555716	1.556471	1.556066	1.555198	1.555993	1.555919	1.558191	1.557352	2018-10-08	2018-10-03

For our final test, let's run the model and see if it can predict the price of Microsoft Corporation, 5 days out (for real!).

```
# create time-series feature engineering
doos = xgb.DMatrix(data = live_stock_ready_df[FEATURES])
# predictions
preds = stock_model.predict(doos)
preds[(len(preds)-10):len(preds)]
```

```
array([1.5438361, 1.5438361, 1.5461913, 1.5461913, 1.5461913, 1.5485295,
       1.5517291, 1.5461913, 1.5485295, 1.5432804], dtype=float32)
```

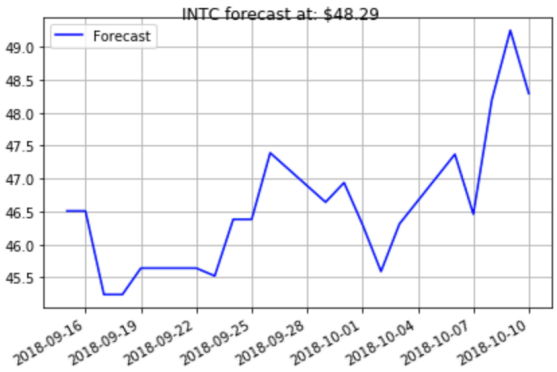
Presenting Forecasts to the Public

Even though we haven't started thinking about any UI or web work, we will need to abstract all this code into clean and easily reusable functions. The first thing we should experiment is how do we want to present the results to our viewers. I selected an arbitrary 20 points of data as that will encompass 15 real points and 5 predicted ones. The end result is a very clear and intuitive chart of

where the market is going to go.

```
# apply the expononent so it's in readable dollar amounts
future_df_tmp['forcast'] = np.exp(list(preds))
# just need a couple of rows
future_df_tmp = future_df_tmp.tail(20)

future_df_tmp = future_df_tmp.sort_values('prediction_date')
fig, axes = plt.subplots()
plt.suptitle(symbol + ' forecast at: $'+
str(np.round(future_df_tmp['forcast'].values[-1],2)))
plt.plot(pd.to_datetime(future_df_tmp['prediction_date'].astype(s
tr)), future_df_tmp['forcast'], color='blue', label='Forecast')
plt.legend()
plt.grid()
plt.xticks(rotation='vertical')
fig.autofmt_xdate()
fig.tight_layout()
plt.show()
```



Abstracting the Code into Logical Functions

(Link to code: https://manuelamunategui.pythonanywhere.com/eBook-code-ml-for-free-code-abstraction.html)

This is the cleaning up stage, we need to package all this information into cohesive functions. Think about this like we do with object-oriented programming. Things need to be packaged into logical containers and be properly named. This will allow us to not only easily move this to the cloud and know where each piece fits, but it will also allow us to test and debug things

when they break. A bit like unit testing, when something is broken, you can call each function individually and ascertain that they each do what they are supposed to do. When one function doesn't behave like it should, you'll know where to focus your debugging attention.

Let's start with our constructor function. The "GetTrainedModel()" only gets called once per server run (i.e. whenever the server is started for the first time or reset).

```
def GetTrainedModel(stock_model_ready_df, TARGET):
    from sklearn.model_selection import train_test_split
    FEATURES = [f for f in list(stock_model_ready_df) if f not in
[TARGET, 'prediction_date', 'last_market_date', 'symbol']]

    x_train, x_test, y_train, y_test =
train_test_split(stock_model_ready_df,

stock_model_ready_df[TARGET], test_size=0.33, random_state=42)

    print('x_train.shape:', x_train.shape)
    print('x_test.shape:', x_test.shape)


    import xgboost as xgb
    dtrain = xgb.DMatrix(data = x_train[FEATURES],  label =
y_train)
    dval = xgb.DMatrix(data = x_test[FEATURES], label = y_test)

    param = {'max_depth':3,
                'eta':0.05,
                'silent':0,
                "objective":"reg:linear",
                "eval_metric":"rmse",
                'subsample': 0.8,
                'maximize': False,
                'colsample_bytree': 0.8}

    evals = [(dtrain,'train'),(dval,'eval')]
    stock_model = xgb.train ( params = param,
                  dtrain = dtrain,
                  num_boost_round = 1000,
                  verbose_eval=50,
                  early_stopping_rounds = 500,
                  evals=evals)
    return(stock_model, FEATURES)
```

```
# load model ready data
stock_model_ready_df = pd.read_csv('stock_model_ready_df.csv')
# set all features to strings
stock_model_ready_df.columns = [str(f) for f in
list(stock_model_ready_df)]

# train the xgb model whenever the server is reset
# You could also train the model in advance and load the saved
version up to skip this step
xgb_stock_model, FEATURES = GetTrainedModel(stock_model_ready_df,
TARGET='outcome')
```

The function takes the data ready for modeling and models it. As I mentioned before this is only done once during a server session so you will notice a little delay when starting things up. It returns a trained model ready to forecast anything our visitors desire.

Next we build two "**live**" functions. These will be called whenever a user selects a stock from the drop-down. "**GetLiveStockData()**" will query our live data repository and download the latest data for that particular stock (see notebook for full version):

```
def GetLiveStockData(symbol, size=50):
    # we'll use pandas_datareader
    import datetime
    pd.core.common.is_list_like = pd.api.types.is_list_like
    import pandas_datareader.data as web

    try:
        start = datetime.datetime(2018, 1, 1)
        end = datetime.datetime.now()
        live_stock_data = web.DataReader(symbol, 'robinhood',
start, end)
        live_stock_data.reset_index(inplace=True)
        live_stock_data = live_stock_data[['symbol', 'begins_at',
'close_price']]
        live_stock_data.columns = ['symbol', 'date', 'close']
    except:
        live_stock_data = None
  …
```

```
    return(live_stock_ready_df)
```

This function takes as input the stock symbol and size to pull. It "**web.DataReader()**" function to download the needed stock data. It will do some data cleaning and transformation to get it in the exact same format and with the same features as our training data.

If we run it with the following seed data, we get:

```
symbol = 'INTC'
stock_ready_df = GetLiveStockData(symbol, size=50)
stock_ready_df.tail()
```

```
currMonth: 10
currYear: 2018
symbol: INTC
```

	0	1	2	3	4	5	6	7	8	9	prediction_date	last_market_date
35	3.815953	3.830813	3.831897	3.854394	3.842887	3.848231	3.826683	3.822098	3.826029	3.856299	2018-10-06	2018-10-01
36	3.830813	3.831897	3.854394	3.842887	3.848231	3.826683	3.822098	3.826029	3.856299	3.838376	2018-10-07	2018-10-02
37	3.831897	3.854394	3.842887	3.848231	3.826683	3.822098	3.826029	3.856299	3.838376	3.873282	2018-10-08	2018-10-03
38	3.854394	3.842887	3.848231	3.826683	3.822098	3.826029	3.856299	3.838376	3.873282	3.886910	2018-10-09	2018-10-04
39	3.842887	3.848231	3.826683	3.822098	3.826029	3.856299	3.838376	3.873282	3.886910	3.873906	2018-10-10	2018-10-05

This confirms that we are getting the latest stock data and transformed into the proper format.

We're almost there, one more abstracted function. We finally write the function that will create the plot. Here I will show you the normal version but we will have to HTML'ize it for the cloud.

```
def GetPredictionChart(symbol, stock_df, xgb_model, FEATURES):
    import xgboost as xgb
    # create time-series feature engineering
    doos = xgb.DMatrix(data = stock_df[FEATURES])
    # predictions
    preds = xgb_model.predict(doos)

    # just pull las 10 preds and build chart
```

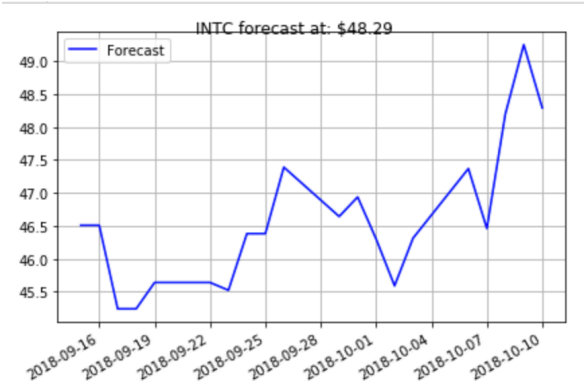
```
    future_df_tmp = stock_df.copy()
    future_df_tmp['forcast'] = np.exp(list(preds))
    # just need a couple of rows
    future_df_tmp = future_df_tmp.tail(20)

    future_df_tmp = future_df_tmp.sort_values('prediction_date')
    plt.suptitle(symbol)
    plt.plot(future_df_tmp['prediction_date'].astype(str),
future_df_tmp['forcast'], color='blue', label='Forecast')
    plt.legend()
    plt.grid()
    plt.xticks(rotation='vertical')
    plt.show()

    return(preds[(len(preds)-10):len(preds)])
```

Very similar to what we were doing earlier. "**GetPredictionChart()**" takes the stock symbol name, the live stock data, the trained XGBoost model, and the feature list. It runs the predictions on the data and builds the plot. We can test it with the following commands:

```
preds = GetPredictionChart(symbol, stock_ready_df,
xgb_stock_model, FEATURES)
```



The Local Web Application

(Link to code: https://manuelamunategui.pythonanywhere.com/eBook-code-ml-for-free-local-flask-application.html)

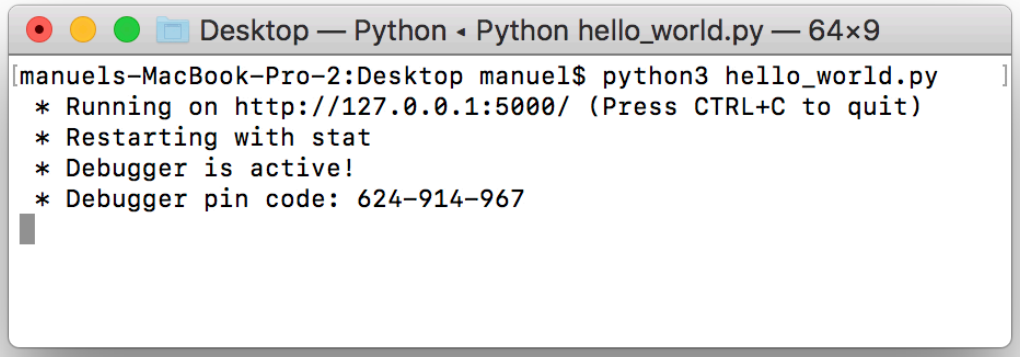
Before diving into our Flask application, I recommend you run a simple local '**Hello World**' example from the official help docs at http://flask.pocoo.org/docs/1.0/quickstart/

Simply create a Python script with the following code, save it to file as "**hello_world.py**". Run it by calling "**python3 hello_world.py**"

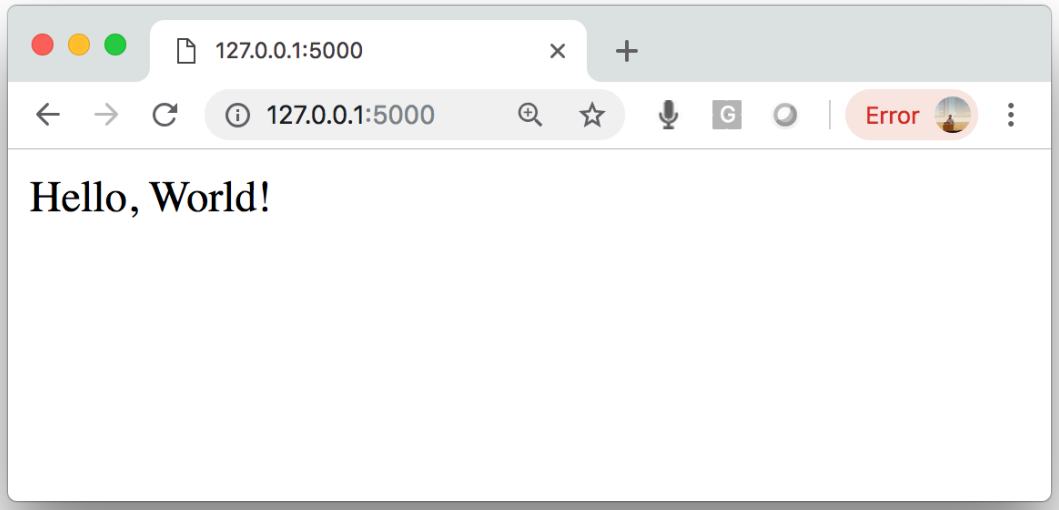
```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!'

if __name__ == "__main__":
    app.run(debug=True)
```



You will see your local host address appear in the command window, paste it into a browser and you will see 'Hello World' appear in the browser.



The typical Flask application folder structure has a controlling Python script, "**templates**" that hold the HTML files that can be controlled dynamically, and a "**static**" folder that mostly holds images. It looks like the following:

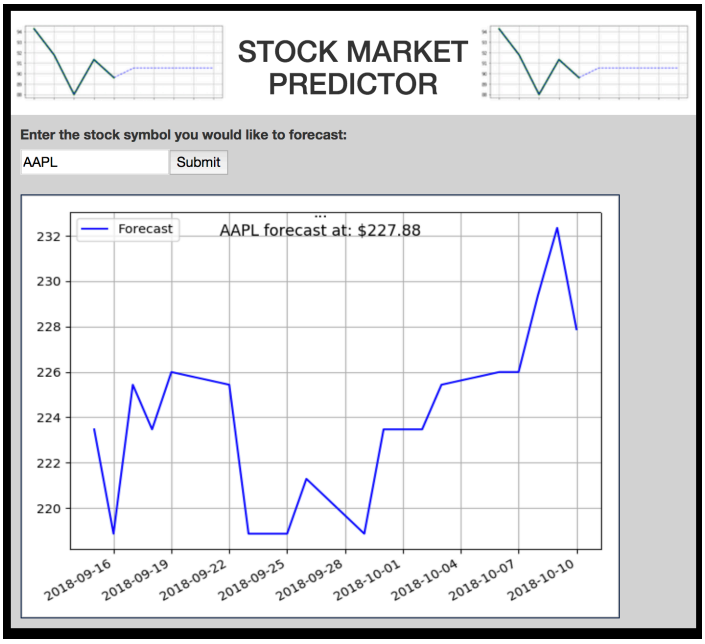
```
/main.py
/templates
      /some_web_page.html
/static
      /images
            /some_image.jpg
```

The Stock Market Predictor

Let's build our own Flask web application and embed our abstracted functions into it. We already know the functionality and look required by our fictional customer (these are all things that need to be figured out before even writing one line of code - the functionality, technical specification, and UI).

Here is a look at our final product when it is first started:

And here is a look at it with a forecast:



So go ahead and create a new folder called '**stock-market-predictor**', and add the following files (you can find the full code for all these files on line, so copy the code and create them using your favorite text/HTML editor).

The files we will add are:

```
/stock_model_ready.csv
/main.py
      /templates
            /show-me-the-future.html
      /static
            /images
                  /forecast-chart.png
```

forecast-chart.png

To get a copy of this image, simply right-click and save the image from the source code file and save it to the '**stock-market-predictor/static/images/**' folder.

main.py

Our "**main.py**" script is the brains of the operation. The WSGI server running on PythonAnywhere will know to call this file to get all its web serving requests fulfilled.

```
@app.before_first_request
def PrepareData():
    global FEATURES, xgb_stock_model, options_stocks

    # load model-ready data
    stock_model_ready_df = pd.read_csv(os.path.join(BASE_DIR,
'stock_model_ready.csv'))
    # set all features to strings
    stock_model_ready_df.columns = [str(f) for f in
list(stock_model_ready_df)]

    # train the xgb model whenever the server is reset
    # You could also train the model in advance and load the
saved version up to skip this step
    xgb_stock_model, FEATURES =
GetTrainedModel(stock_model_ready_df, TARGET='outcome')


def GetTrainedModel(stock_model_ready_df, TARGET):
   …

  return(live_stock_ready_df)


def GetPredictionChart(symbol, stock_df, xgb_model, FEATURES):
    …

    # this is the part that we change to pass the HTML-converted
image instead of calling plt.show()
    img = io.BytesIO()
    plt.savefig(img, format='png')
    img.seek(0)
    plot_url = base64.b64encode(img.getvalue()).decode()
```

```
    chart_plot = Markup('<img style="padding:1px; border:1px
solid #021a40; width: 90%; height: 90%" src="{}">'.format(plot_url))
    return(chart_plot)


@app.route('/', methods=['POST', 'GET'])
def get_financial_information():
    chart_plot = ''
    selected_stock = ''

    if request.method == 'POST':
        selected_stock = request.form['selected_stock']

        if (len(selected_stock) > 0):

            # get latest data
            currMonth = datetime.now().month
            currYear = datetime.now().year
            stock_df = GetLiveStockData(selected_stock,
currMonth, currYear)
            if (stock_df is not None):
                if len(stock_df) > 0:
                    chart_plot =
GetPredictionChart(selected_stock, stock_df, xgb_stock_model,
FEATURES)

    return render_template('show-me-the-future.html',
        selected_stock = selected_stock,
        chart_plot=chart_plot)
```

How It Works

Each Flask-specific function has a decorator above it. For example, "**@app.before_first_request**" is the function the Flask server will call before it processes any requests. Basically, this is a function that gets called only once when the server is started or restarted. If you look at the above code, this decorator calls the "**PrepareData()**" which will load the stock data, do some minor data preparation, then call the "**GetTrainedModel()**" to train the market forecasting model. Keep in mind that even-though this operation takes a little time, it is only performed once during each server session.

After that call, the web server stays in listening mode until someone calls the web site in question, in our case the local host. When someone calls the root URL, the "**@app.route('/', methods=['POST', 'GET'])**" directs the request to the "**get_financial_information()**" function. The first time around, a black page is shown with an input text box, once the user inputs a stock symbol, the same function will notice that the user has passed some custom information and our function needs to get to work as we did with the stand-alone script. The big difference here from the stand-along version is that it has to pull the stock symbol from the HTTP Post "**request.form['selected_stock']**". From then on everything is the same.

Once the data is pulled and the chart is produced, another difference is the way the financial stock chart is generated. If you look at the "**GetPredictionChart()**" function, around the end you will notice the following snippet of code:

```
image instead of calling plt.show()
img = io.BytesIO()
plt.savefig(img, format='png')
img.seek(0)
plot_url = base64.b64encode(img.getvalue()).decode()

chart_plot = Markup('<img style="padding:1px; border:1px solid
#021a40; width: 90%; height: 90%" src="
{}">'.format(plot_url))
```

This is where things take a new direction as I alluded earlier. We need to pass the stock chart in a dynamic format, not as a saved file. Imagine if you had ten simultaneous users and each created an image on the server, you wouldn't know which image went where. This is why we use the "**base64.b64encode()**" function as it will create a text-based image that a browser can dynamically interpret as an image.

If you run the code and look at the source code while a stock chart is visible you will see the following:

```
img style="padding:1px; border:1px solid #021a40; width: 90%;
```

```
height: 90%" src="
QICAgIfAhkiAAAAAlwSFlzAAAPYQAAD2EBqD+naQAAADl0RVh0U29mdHdhcmUAbWF
0cGxvdGxpYiB2ZXJzaW9uIDIuMi4yLCBodHRwOi8vbWF0cGxvdGxpYi5vcmcvhp/
UCwAAIABJREFUeJzs3XlcVPX+P/
DXDAzDJiIgIMqiYIqKuKW5pJiKimGLy9XMpavmz6+2uKR5K9OyXFs0TW+3W1rXSi0
zMzNwS1OvuWFdDMQFcUNSgFYTi/
P05nYGRAljPreT0fDx4znDlzzufDqLx8fz7nc1SCIAggSCIAggIiIiIsVQW7sBRERERGRZD
IBEREREsMASERERKQwDIBEREREsMASERERKQwDIBEREREsMASERERKQwDIBEREREsM
ASERERKQwDIBEREREsMASERERKQwDIBEREREsMASERERKQwDIBEREREsMASERERKQwDIBEREREsM
ASERERKQwDIBEREREuNs7QZYkiAIuH37trWbQURERFQn…
```

That’s a portion of a stock chart that can be passed from the web server to the client machine and displayed accordingly. Nothing is saved to file and each client session automatically generates their own and consumes it immediately.

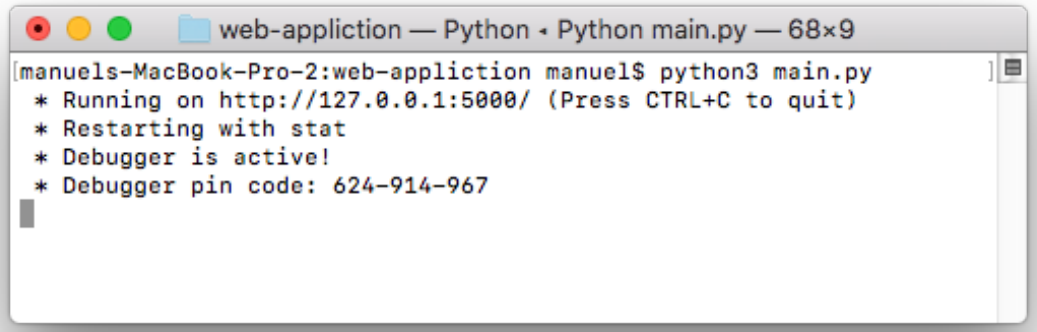
show-me-the-future.html

Now let’s take a look at the HTML template. This is the web page that will ask the user for a stock symbol and then display the forecast. Notice the below snippet which contains a few double curly brackets. This is how Flask injects variables into the HTML code using Jinja2 (http://jinja.pocoo.org/docs/2.10/).

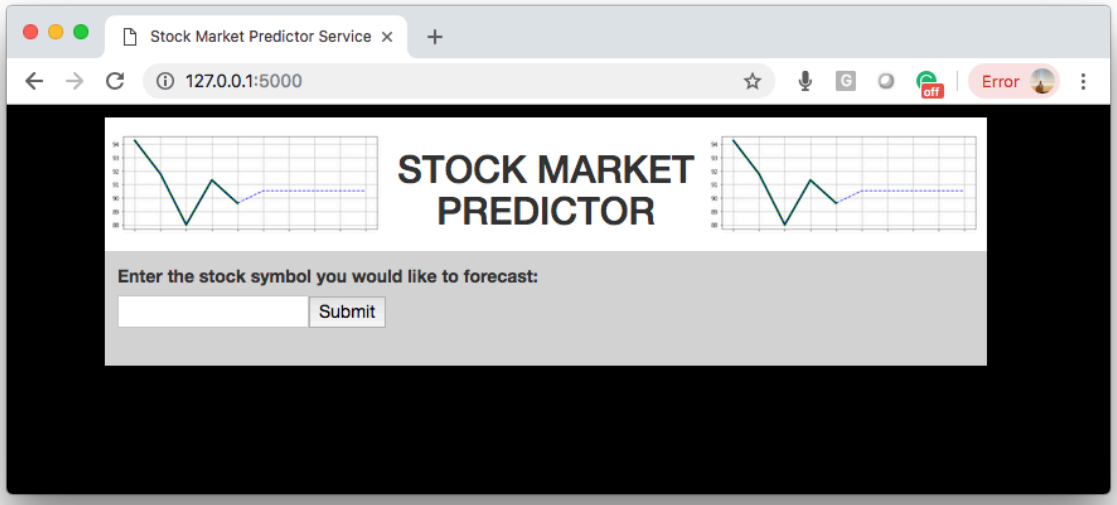
```
      <table border=0 cellpadding="10" style="width: 700px;
background-color:lightgrey;">
        <tr>
          <td colspan='2'><label>Enter the stock symbol you would
like to forecast:</label>
            <FORM id='submit_content' method="POST"
action="{{ url_for('get_financial_information') }}">
              <input type="text" name="selected_stock"
value="{{selected_stock}}"><input type="submit" value="Submit">
            </FORM>
          </td>
        </tr>
        <tr>
          <td>{{chart_plot}}</td>
        </tr>
      </table>
```

Running the Local Version

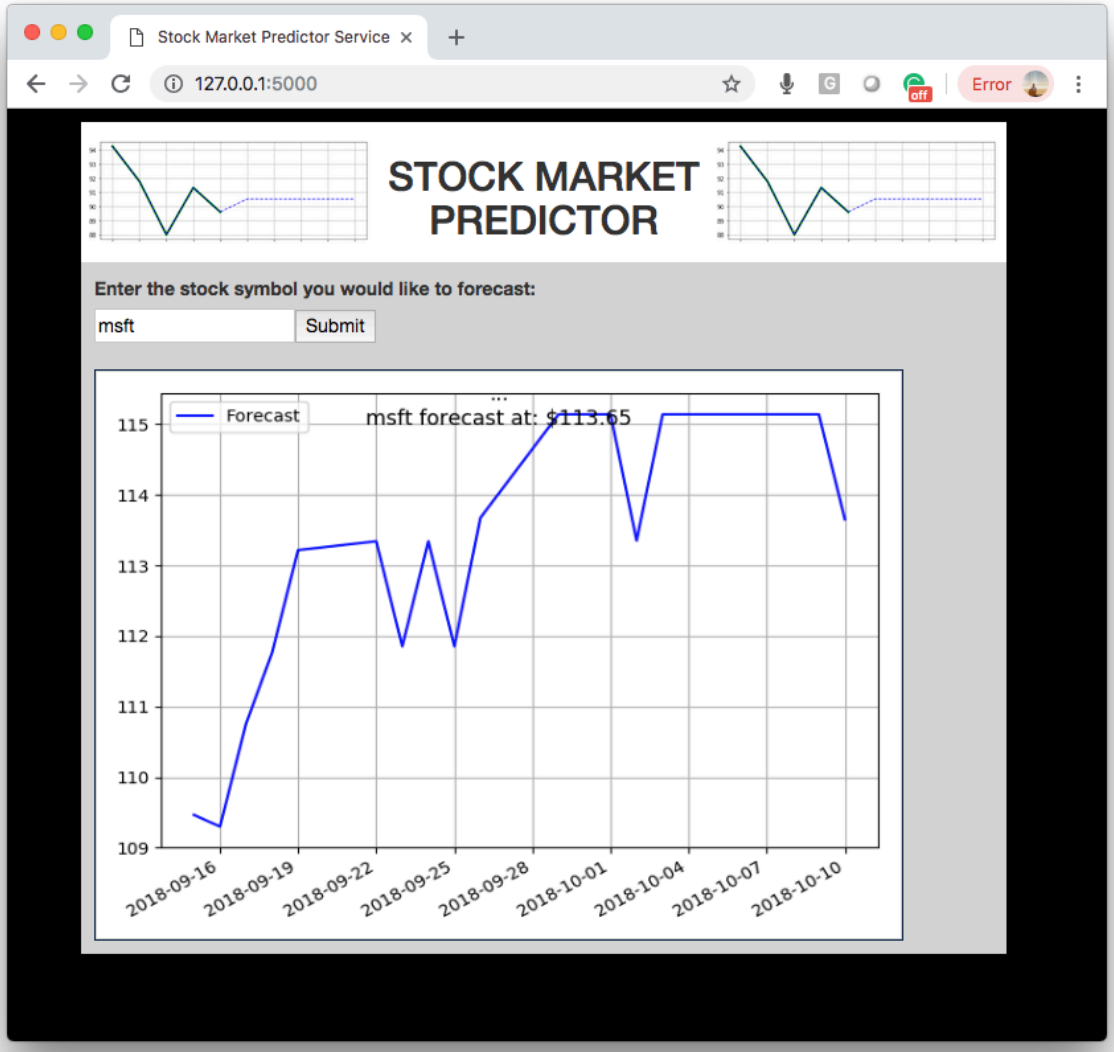
Just like we did with the "**hello_world.py**" example. You now need to open a command window and call: "**python3 main.py**":



And then take that local address and paste it into a web browser. Keep in mind that it may take a little while to show the page as it first needs to prepare the data and run the XGBoost model.



Take it through its paces to make sure it works well and that it handles non-existent stocks (you can also extend it to add more thorough error handling).

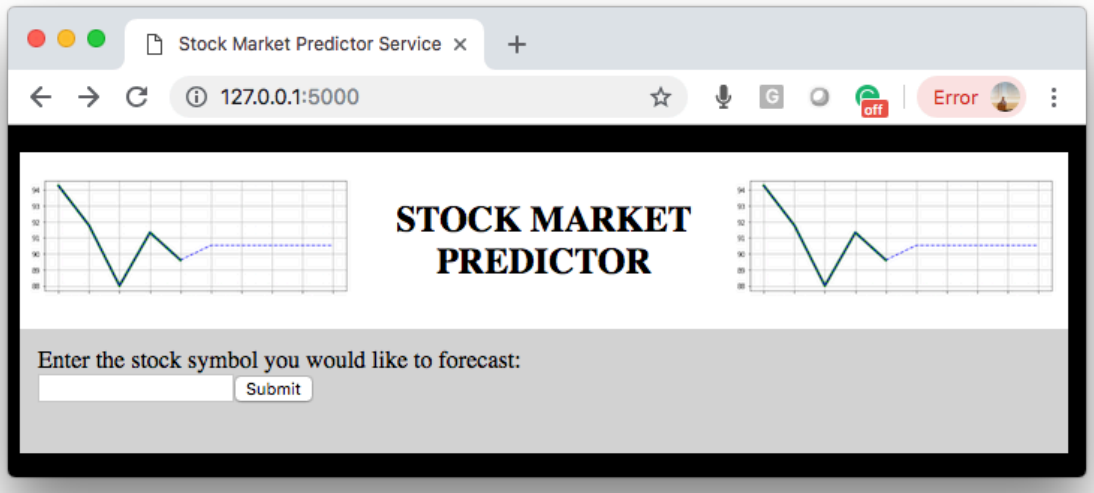

Stock Market Predictor Service
127.0.0.1:5000
Error
STOCK MARKET PREDICTOR
Enter the stock symbol you would like to forecast:
msft
Submit
Forecast
msft forecast at: $113.65
115
114
113
112
111
110
109
2018-09-16
2018-09-19
2018-09-22
2018-09-25
2018-09-28
2018-10-01
2018-10-04
2018-10-07
2018-10-10

Bootstrap

Bootstrap is great! It will make your website look professional without much work - just add a few "**link**" and "**scripts**" tags to your HTML pages and you will inherit, for free, that professional look that most of the other sites have out there (and they're probably using the same trick).

It is used by almost 13% of the web according to BuiltWith Trends (https://trends.builtwith.com/docinfo/Twitter-Bootstrap). It contains all sorts of great looking styles and behavior for most web tags and controls. By simply linking your web page to the latest Bootstrap CSS will give any boring HTML page an instant and professional-looking makeover. And by putting a little work in using the right templates and tags you can make you site look phenomenal (and we will leverage Bootstrap in each of our web applications).

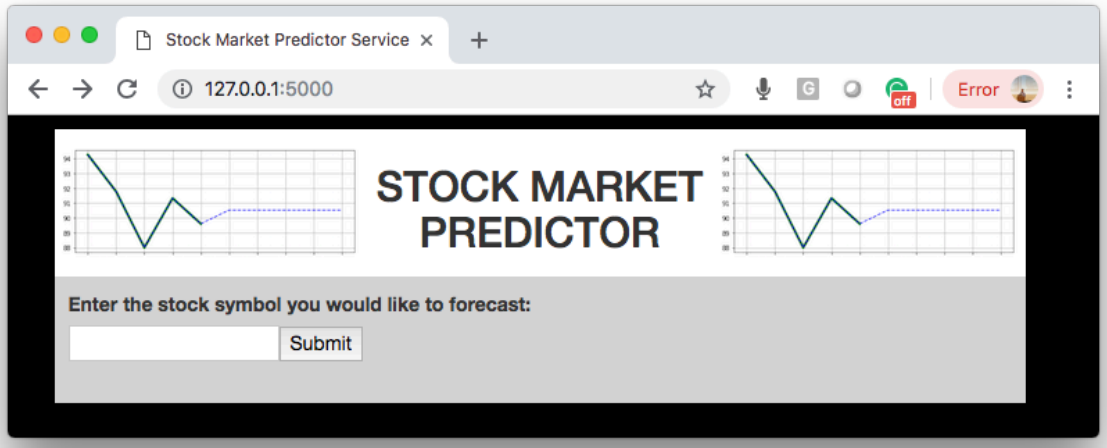
If you don't add Bootstrap, this is what our front end form would look like:



But, by adding the following Bootstrap custom style sheets (CSS) files and some Java Script:

```
   <script src="//ajax.googleapis.com/ajax/libs/jquery/3.3.1/
jquery.min.js"></script>
  <link rel="stylesheet" href="//netdna.bootstrapcdn.com/
bootstrap/3.0.3/css/bootstrap-theme.min.css">
   <link rel="stylesheet" href="//netdna.bootstrapcdn.com/
bootstrap/3.0.3/css/bootstrap.min.css">
   <script src="//netdna.bootstrapcdn.com/bootstrap/3.0.3/js/
bootstrap.min.js"></script>
```

We get this (look closely at the fonts and at the button). It makes for a subtle difference but if it was a more complicated site, the difference would be huge. And all we did was add some basic CSS files…



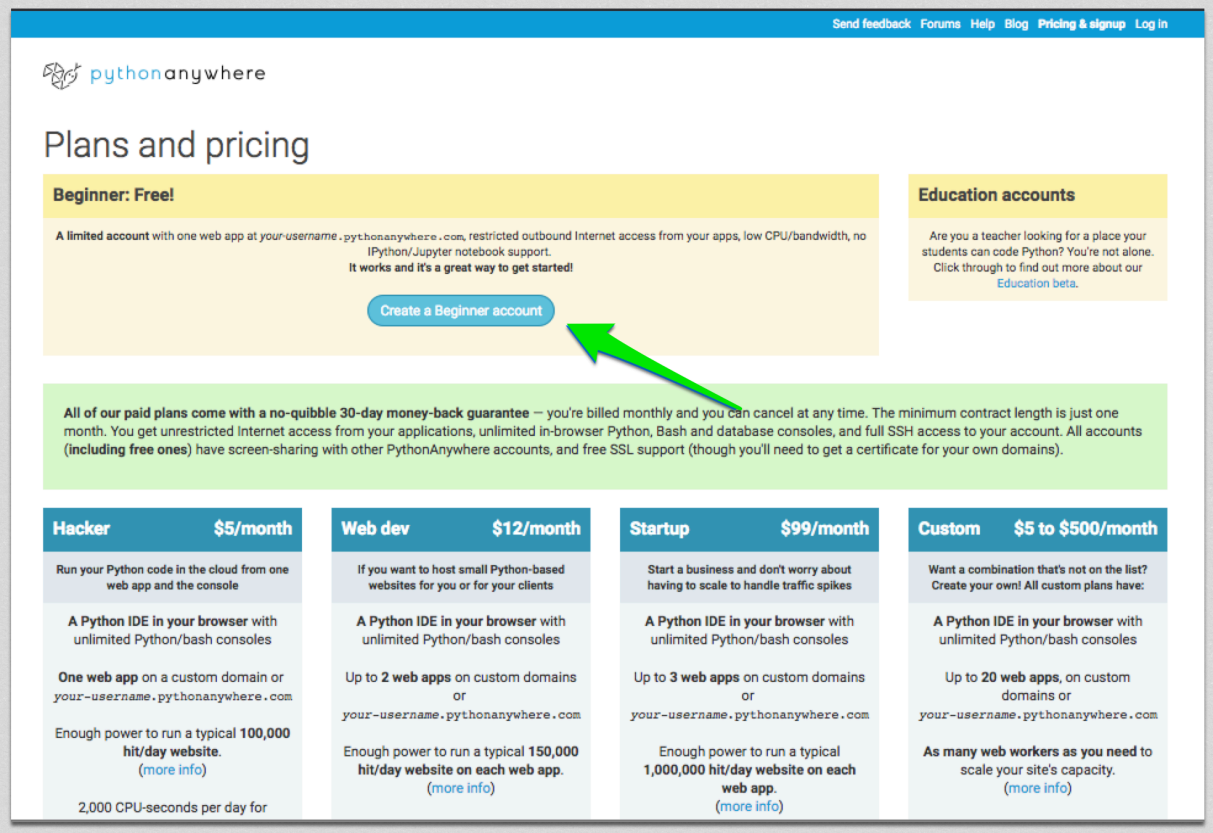
For additional information and training on Bootstrap, check out the official documents on GetBootstrap.com.

To The Cloud and PythonAnywhere

(No code for this chapter)

Let's port our web application to the cloud using PythonAnywhere. This is a very exciting milestone as soon, the entire world will be able to enjoy all your hard work.

The first thing you need to do is get a python anywhere account. All you need is the free-tiered account.

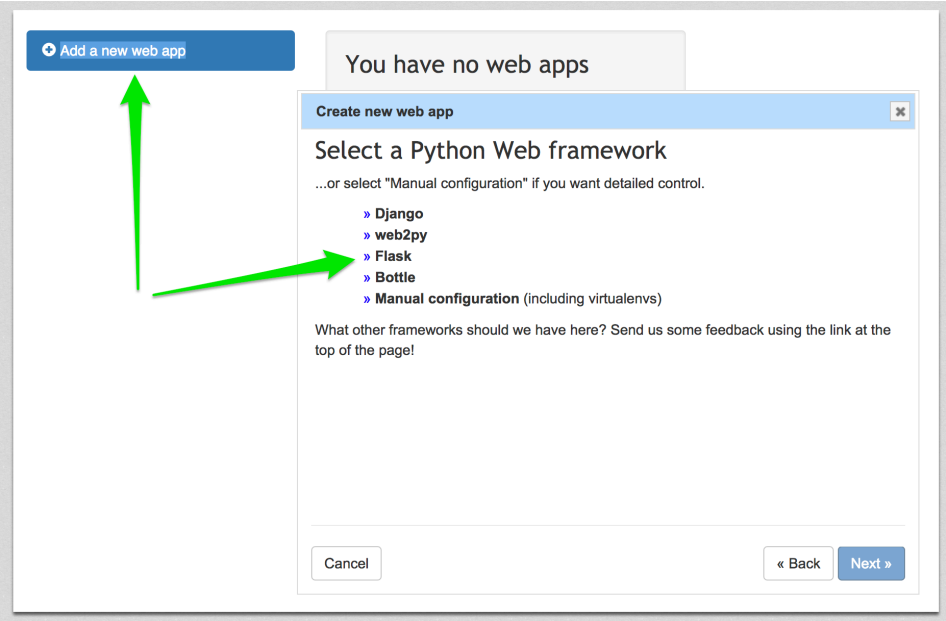


Once you've signed up, we will create our web application using the wizard and run through the hello world example. It's actually super easy to do as the wizard automatically creates one when you set it up.

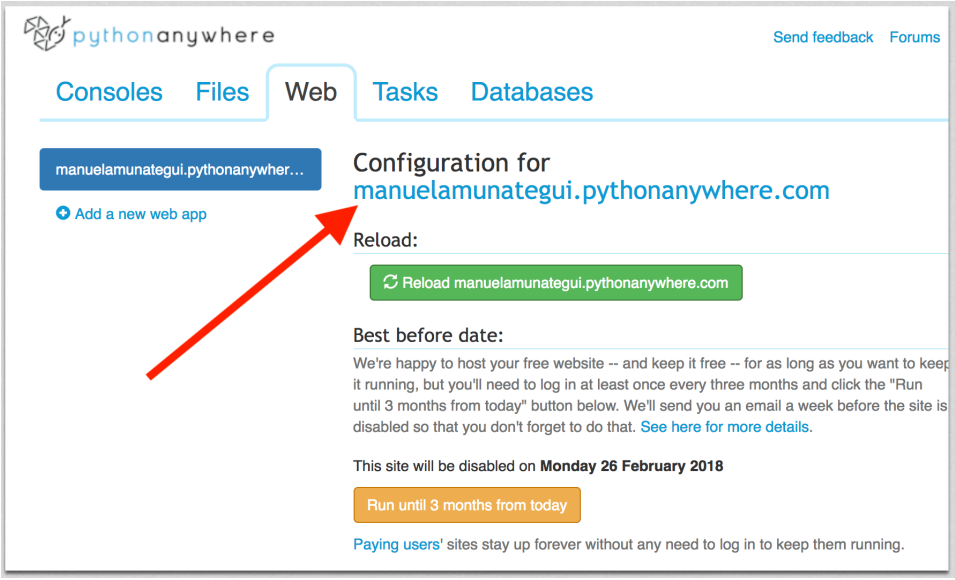
Setting Up Flask Web Framework

Under the '**Web**' tab, click the '**Add a new web app**' blue button. And accept the

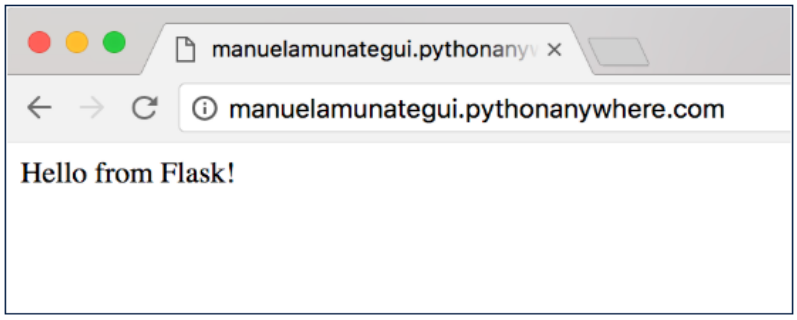
defaults until you get to the 'Select a Python Web framework' and click on '**Flask**' and then the latest Python framework.



You will get to the landing configuration page, hit the green '**Reload your account.pythonanywhere.com**' button (keep in mind yours will have your handle instead of mine) and take your new URL for a spin:



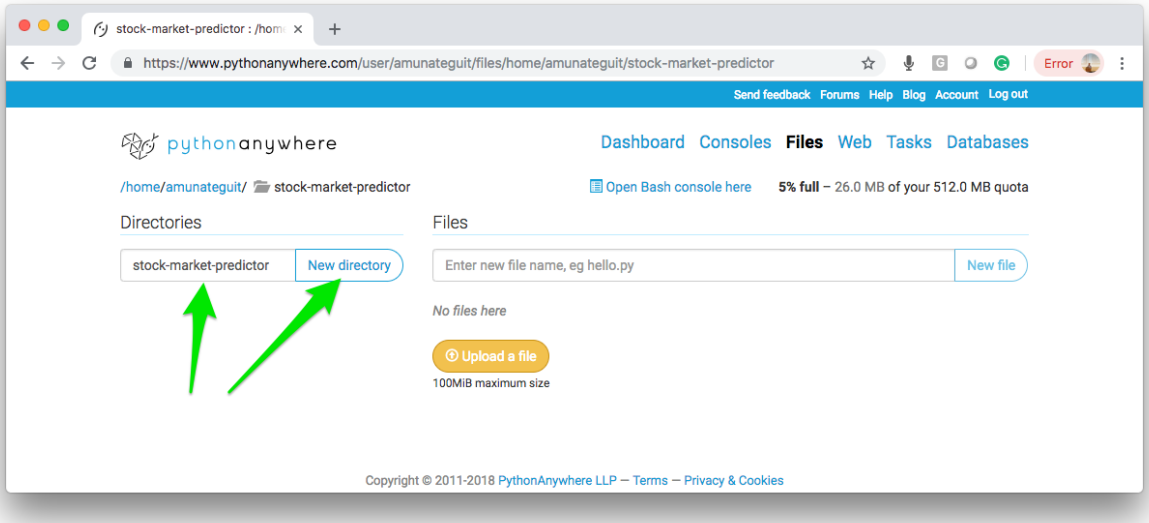
You should see a simple but real web page with the 'Hello from Flask!' message:



PythonAnywhere's "**Hello from Flask!**" is very similar to our own local example earlier.

Uploading the Stock Market Predictor

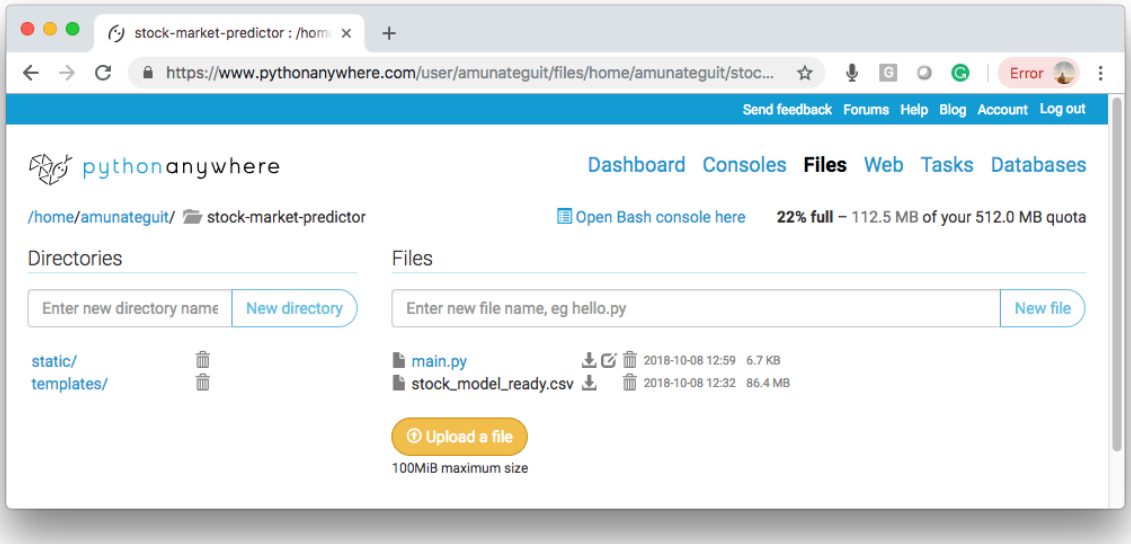
Now it's time to upload our Stock Market Predictor code. This is also very easy to do. We simply need to create a new directory up in the cloud and replicate exactly the folder structure we had on our local machine. So, click on the '**Files**' tab and create a new folder called '**stock-market-predictor**':



Next upload all the folders and files exactly like we did before:

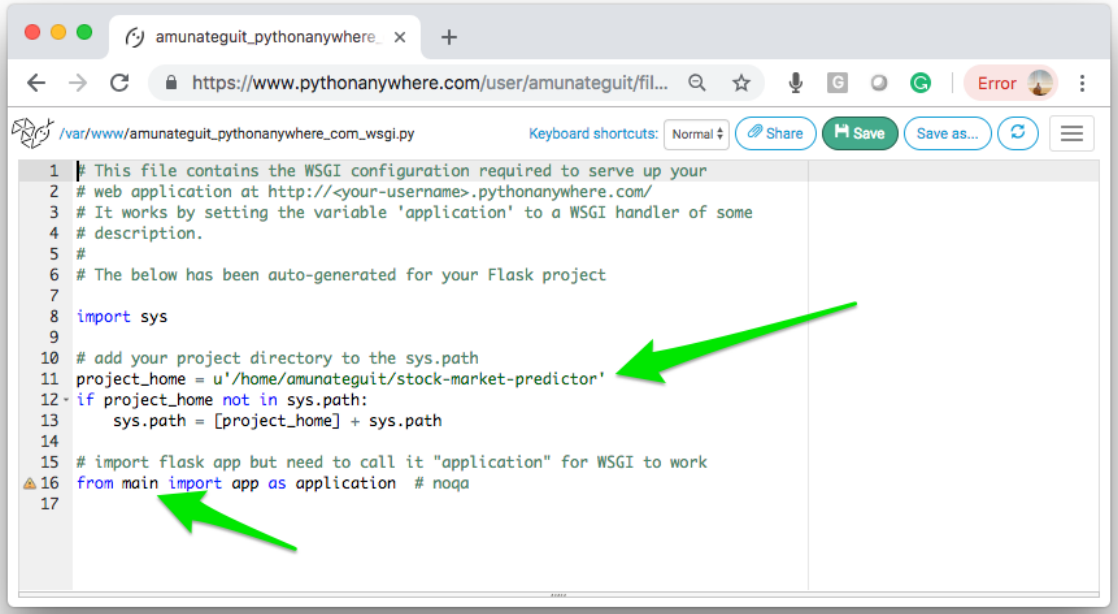
```
/stock_model_ready.csv
/main.py
      /templates
            /show-me-the-future.html
      /static
            /images
                  /forecast-chart.png
```

Your PythonAnywhere account should look something like:



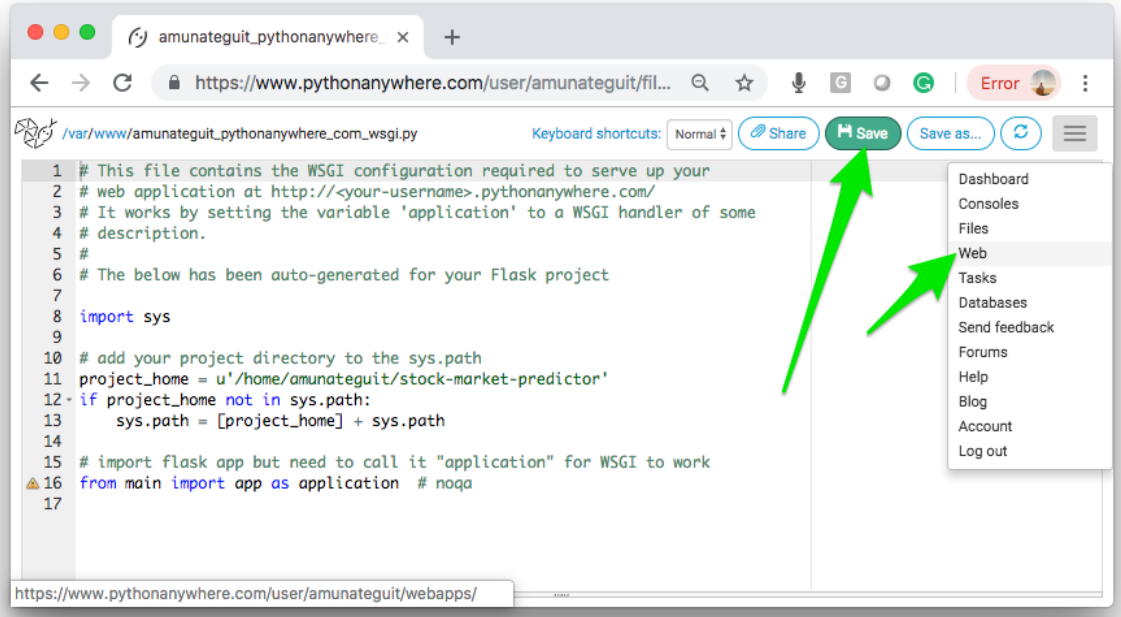
Updated The WSGI Server

We also have to update the WSGI, the Web server, to point to our new main.py, our Flask/Python controlling script. So click on the '**Web**' tab and scroll down to '**WSGI configuration file:/var/www/amunateguit_pythonanywhere_com_wsgi.py**'. Yours will have your own handle, not mine.

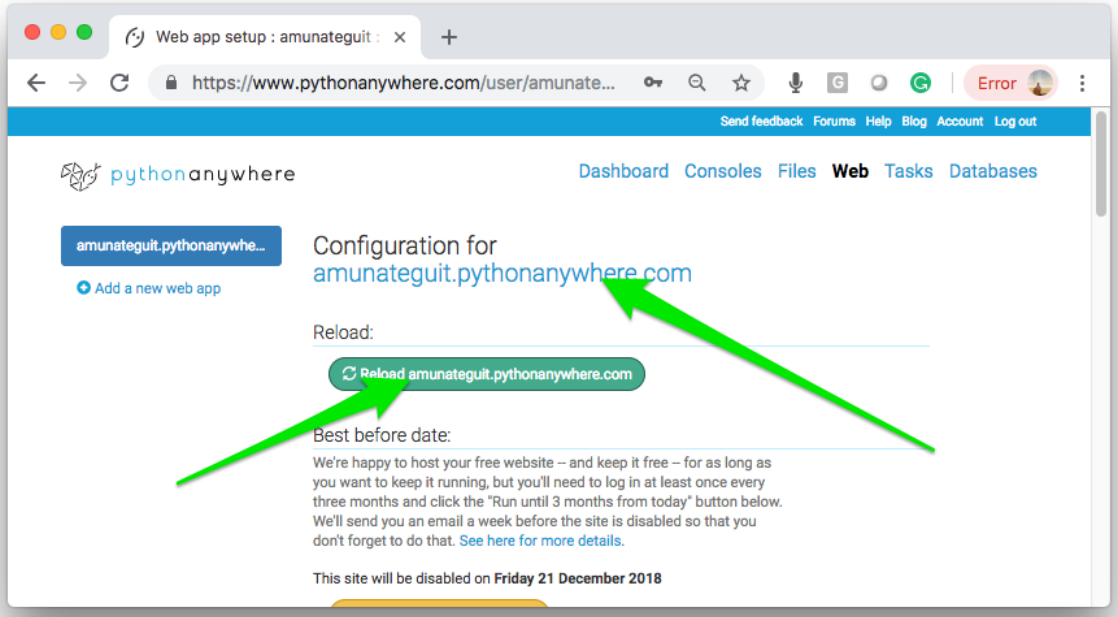


Update line 11 and line 16. Line 11 needs to include your path to the **'stock-market-predictor'** and 16 needs to show '**main**' (as in main.py) and not '**flask_application**' or anything else.

Save the changes and click navigate back to the '**Web**' tab.



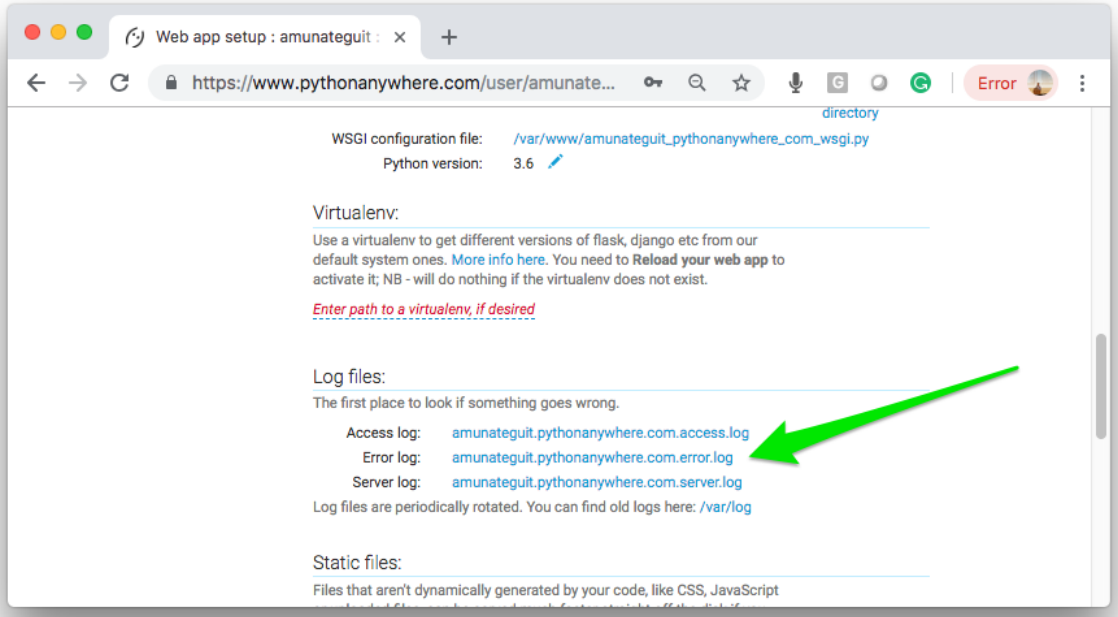
You're almost there. You need to navigate to a '**Web**' server tab, and hit the big green button. Basically, this tells the web server to reboot itself and restart and code that needs to be constructed only once. This means that the data preparation and the XGBoost model are going to be called so be patient and give it a few minutes until the model is trained and saved in memory.



Next click the link right under '**Configuration for…**' just like we did for the '**Hello World**' example before. If, after a few minutes of loading, you see the '**Stock Market Predictor** ', then things are looking good! Make sure you test it out there thoroughly to get through its paces. Our web application is ready for the world.

Troubleshooting

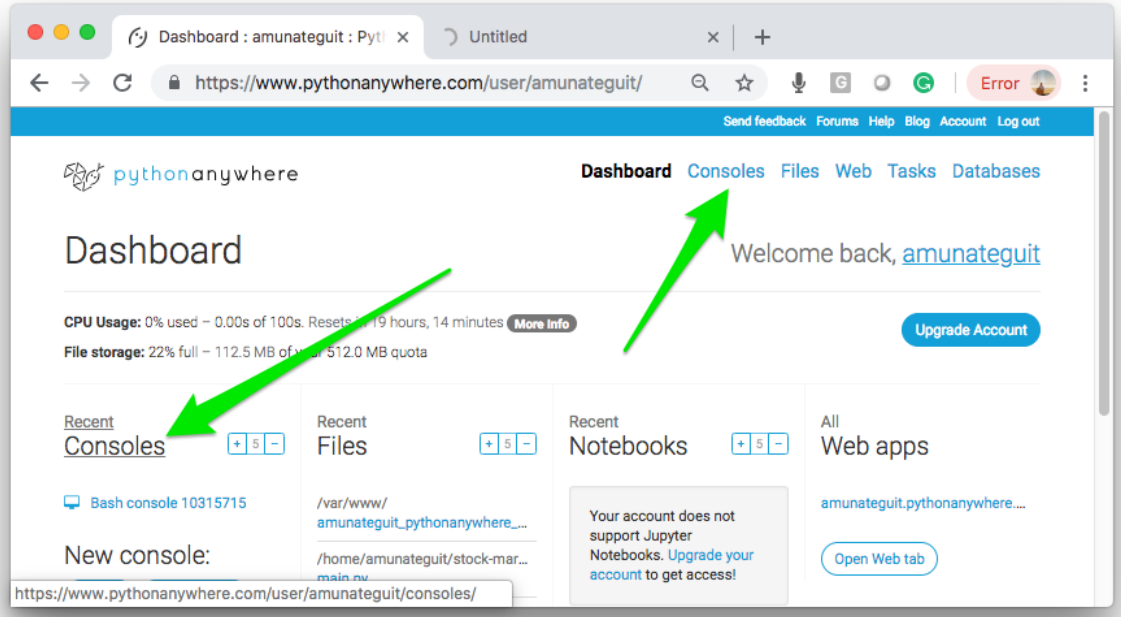
Unfortunately, things don't always go as planned. If you are seeing an error after giving it a few minutes and refreshing it a couple of times, you need to check the error logs. The logs are found on the same '**Web**' tab but a little lower down the page.



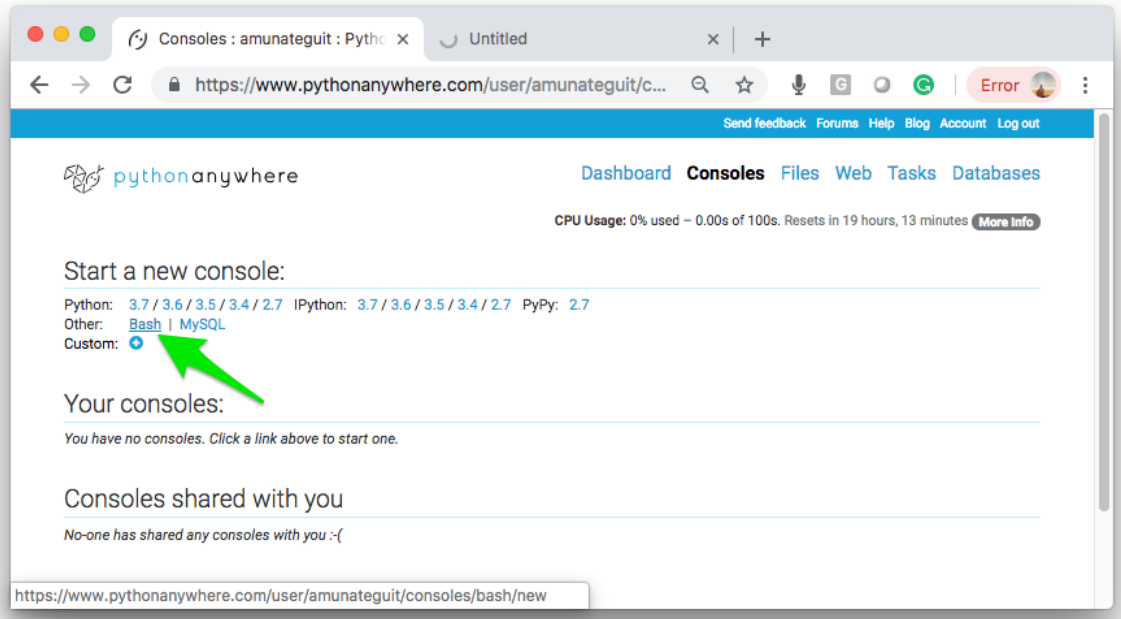
Open it up and see what it is complaining about. You may be missing some Python libraries and that is very easy to address.

Installing Missing Python Libraries

There are a couple of ways of tackling this, see the screenshot below.

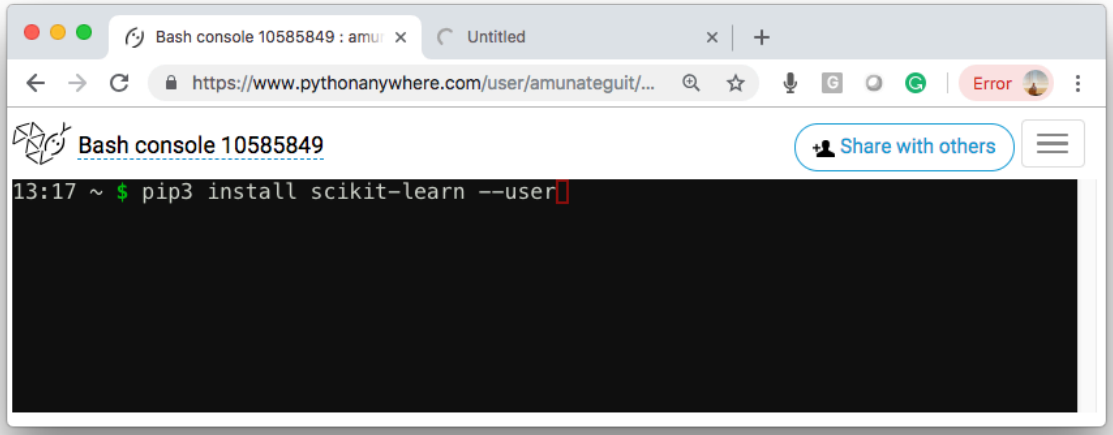


Then click on '**Bash**'



This will open a familiar command-like window. There you can install whatever files the server was complaining about. You will not be able to '**sudo**' install as you are on a shared server, instead, simply append the '**—user**' parameter that

will give you the necessary rights.



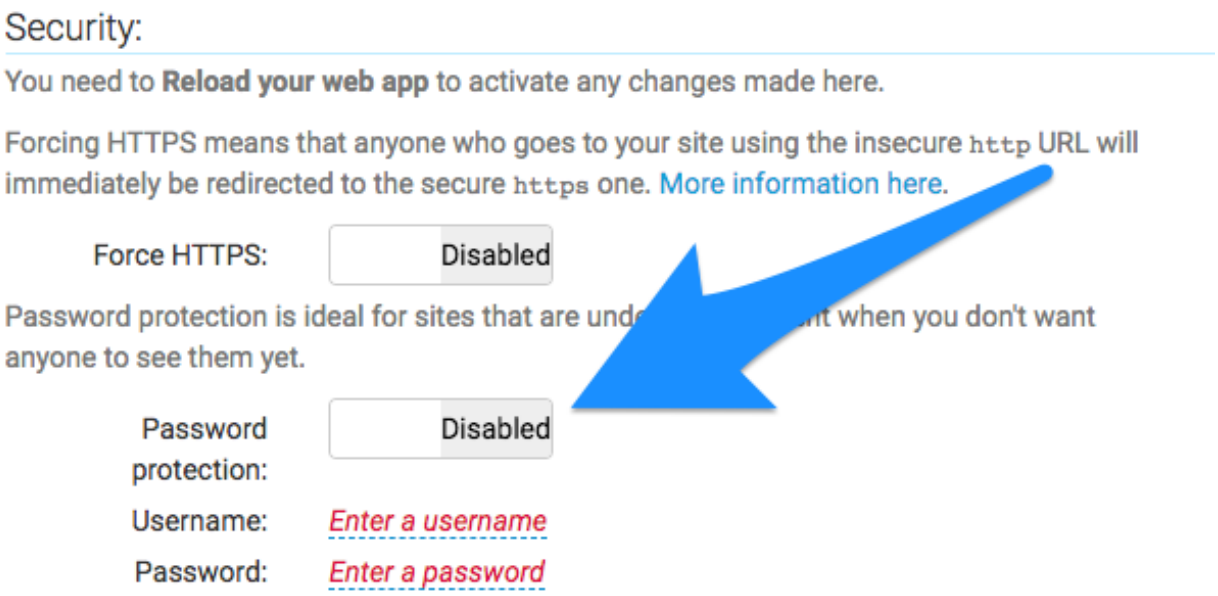
Sell a Product with Stripe

(Link to code: https://manuelamunategui.pythonanywhere.com/eBook-code-ml-for-free-stripe.html)

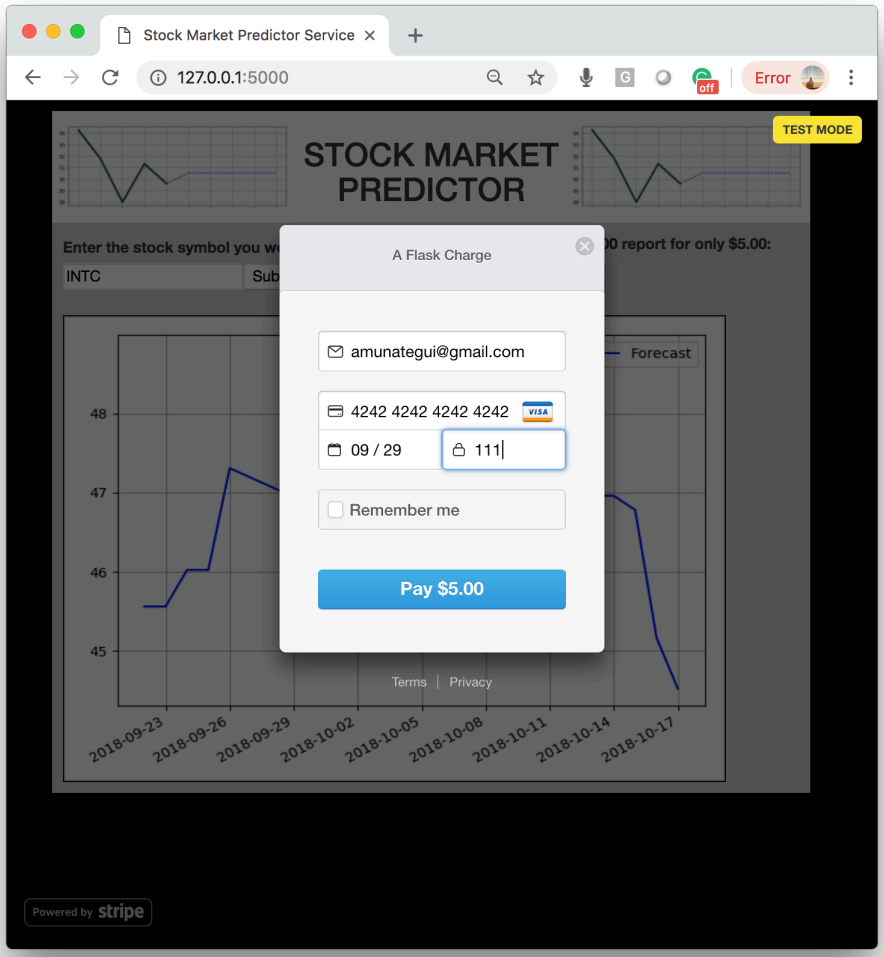
Stripe allows you to take credit card payments without requiring a paid account. It simply takes a percentage directly out of each transaction. This is a great approach, especially if you're not sure whether your idea will work. The transaction cost may be higher than with a paid plan, but it is risk-free for trying new ideas. Remember, we're targeting POC's and prototypes and using riskless tools is the way to go. When things don't work, you simply take the site down and move on to the next idea.

This approach assumes you have a report to sell. Wait for a confirmed purchase email from Stripe then email it to them. You can also send them the URL to a secret web application with added features or to another site (you can create as many free PythonAnywhere accounts as you want). You can even add a password on the site, check out the '**Web**' tab in PythonAnywhere for an easy

option to lock down an entire site:



Remember these are all hacks and temporary solutions to test out POCs. When you conclude that your idea is a winner, you can then properly upgrade the site with the help of a web developer. You'll then have to build a real monetization solution around it. For more information there check out my book [Monetizing Machine Learning: Quickly Turn Python ML Ideas into Web Applications on the Serverless Cloud](), where we explore a real subscription system with Memberful and Stripe.

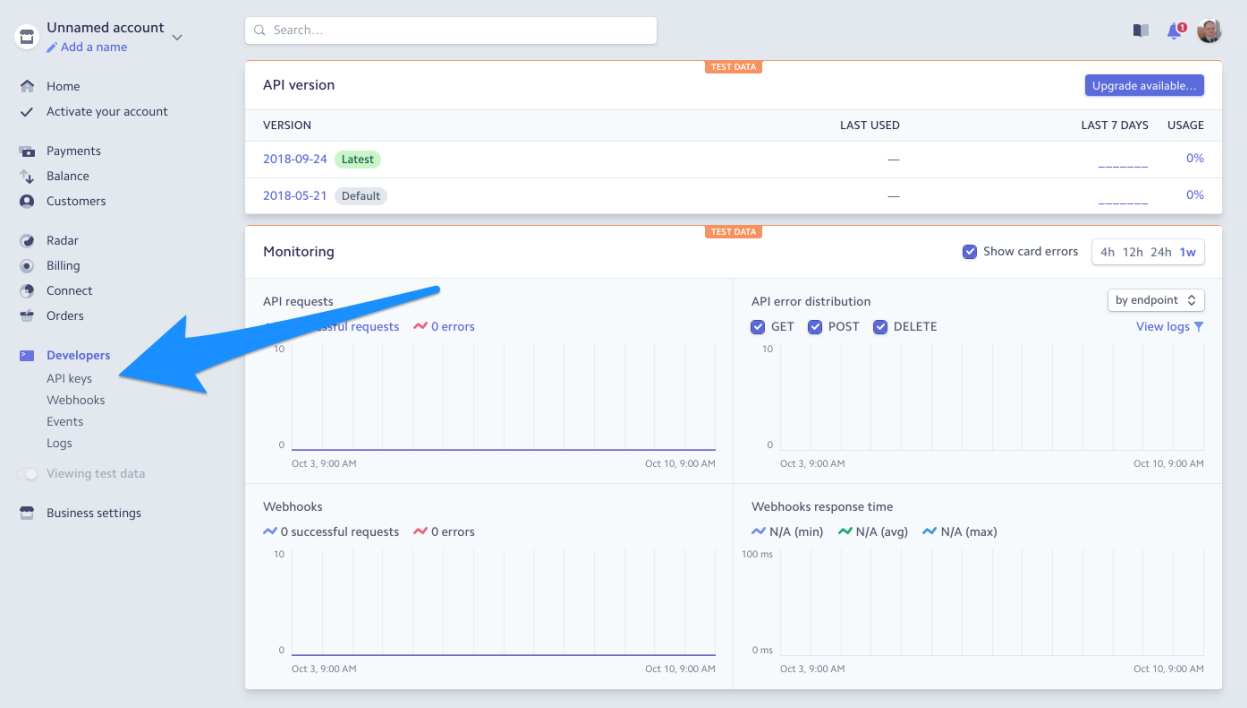


Setting Things Up

Stripe is a simple, powerful, and widely used payment platform with a lot of Flask support. It is also widely trusted, which is important if you want visitors to give you money. We will loosely follow the example from the official docs (https://stripe.com/docs/checkout/flask).

First, sign up for a free testing account which won't require any payment information on your part if you only want to test it out using the developer test account.

Navigate to the "**Developers**" section and click on "**API Keys**"



Get the API keys created for your account as shown in the above screen grab and copy them into the "**main.py**" file under:

```
PUBLISHABLE_KEY='<< YOUR_ PUBLISHABLE_KEY_HERE>>'
SECRET_KEY='<< YOUR_ SECRET_KEY_HERE>>'
```

Code Changes

We'll go through this exercise using a local Flask application as the concept would be exactly the same in the cloud. So, go ahead and download the "**main.py**" and "**show-me-the-future.html**", place them in a similar folder structure as the original version (or replace the old files). Your file structure should look as follows:

```
/stock_model_ready.csv
/main.py
      /templates
```

```
      /show-me-the-future.html
/static
      /images
            /forecast-chart.png
```

Installing Needed Libraries

You will need to install stripe:

```
$ sudo pip3 install --upgrade stripe
```

And take it for a spin:

```
$python3 main.py
```

What's Going on Here?

There really isn't much going on here (well there is but Stripe does all of it), the Python script instantiates a stripe object and feeds it the "**SECRET_KEY**". It also passes the "**PUBLISHABLE_KEY**" as a variable in the "**render_template**" function call. Once the user pushes the "**Pay with Card**" button, a Stripe popup appears and handles the credit card info and email address. This is great as it removes all the security of handling financial information. Instead, Stripe will return a "**stripeToken**" so you can finalize the order by updating the session's strip object. As a confirmation, we pull the email address from the popup Stripe box and add a thank you message. That's it!

```
email_address = request.form['stripeEmail']
message = "Thanks for the order " + email_address + '!'

customer = stripe.Customer.create(
    email=email_address,
    source=request.form['stripeToken']
)

charge = stripe.Charge.create(
    customer=customer.id,
```

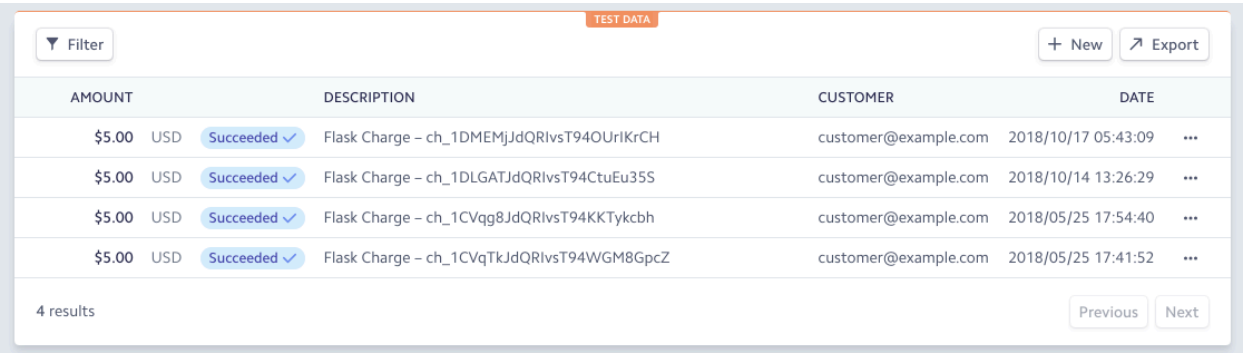
```
        amount=request.form['amount'] ,
        currency='usd',
        description='purchase of SP500 report'
    )
```

In "**show-me-the-future-stripe.html**", we added a simple form script to show the button and handle the transaction submission. The form takes in the "**PUBLISHABLE_KEY**", the dollar amount and description. Sends the whole transaction to a Stripe server and returns to the original page. At that point you will need to handle the transaction once you get the email confirming the purchase (something that can be done manually when prototyping concepts).

```
<form action="/" method="post">
 <article>
    <label>
      <span>Buy the full S&P500 report for only $5.00:</span>
    </label>
  </article>

  <script src="https://checkout.stripe.com/checkout.js"
class="stripe-button"
          data-key="{{ key }}"
          data-description="A Flask Charge"
          data-amount="500"
          data-locale="auto"></script>
</form>
```

If all goes through, you can log back into your Stripe dashboard and confirm the order made it. Pretty easy and low risk for us weekend entrepreneurs!!



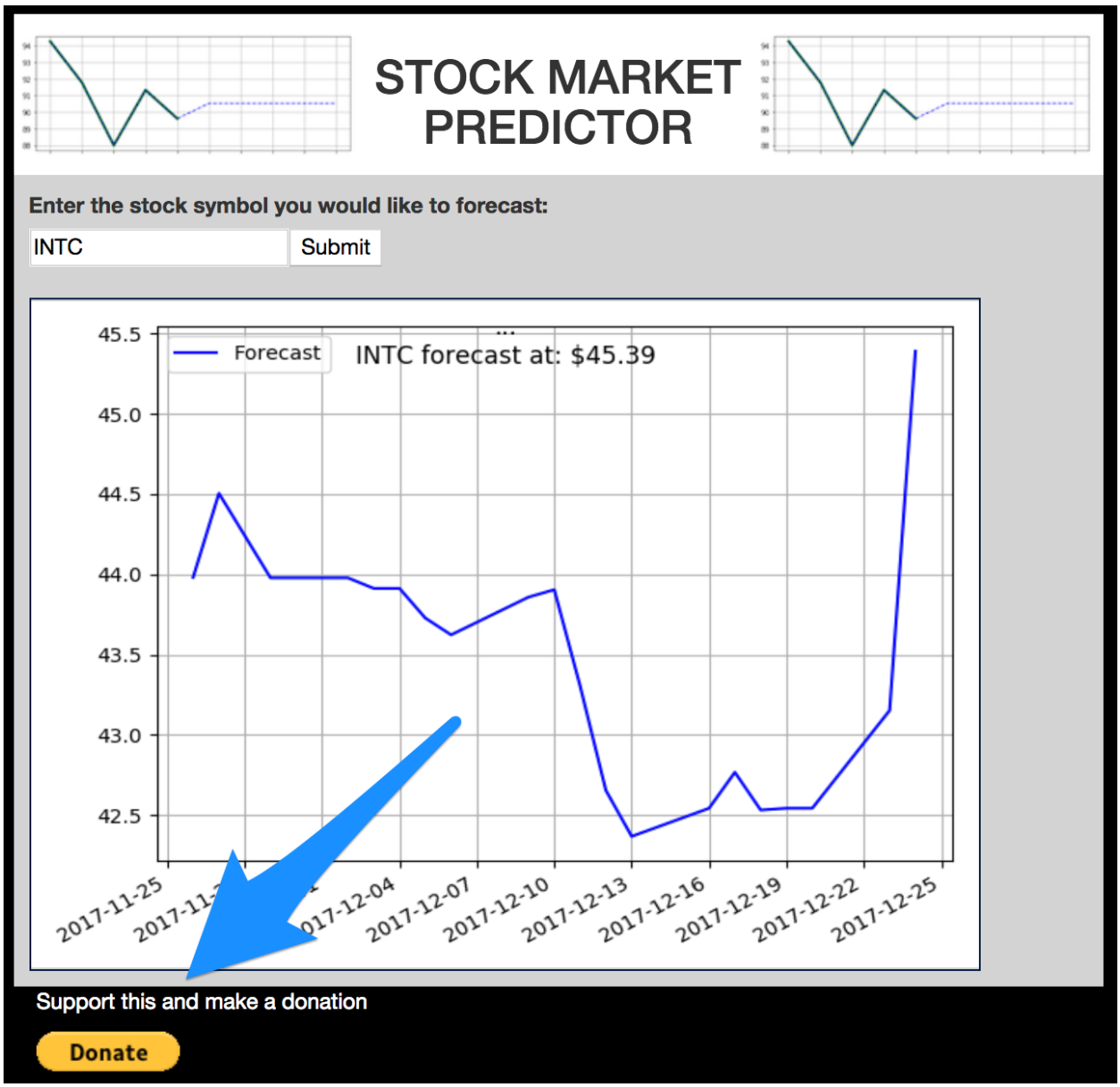
For more information on Stripe, refer to the official docs:
https://stripe.com/docs/checkout/flask

Get Donations with PayPal

(Link to code: https://manuelamunategui.pythonanywhere.com/eBook-code-ml-for-free-paypal-flask-application.html)

Another interesting option that I used in the past, is the PayPal donation button. As long as you have a PayPal account in good standing, you should be able to set this up in under two minutes. So, if what you are offering isn't something that can be sold at a fixed price, or if you think people will appreciate it enough to volunteer a payment, a PayPal donation button may be the next best thing to gauge conversion and monetization potential (check PayPal's buttons page for the fine print as to what is and isn't appropriate for donations).

This was a few years ago but I did use such button and did get donations. I even had one person make a $100 donation - a big thanks for that one!



This is a painless option where you don't even need Flask, as this payment option just requires HTML—nothing else. Go to www.paypal.com/buttons. To get this going, log into your account, select the "**donations**" button, and copy/paste the HTML forms code wherever you want it to appear in your web page. That's it, PayPal will even display the button for you.

Code Changes

We'll go through this exercise using a local Flask application as the concept

would be exactly the same in the cloud. So, go ahead and download the "**main.py**" and "**show-me-the-future.html**", place them in a similar folder structure as the original version.

```
/stock_model_ready.csv
/main.py
      /templates
            /show-me-the-future.html
      /static
            /images
                  /forecast-chart.png
```

Once a visitor clicks on the donation button, PayPal will take it from there and follow-through with the transaction (you will get an email once the payment goes through).

```
<form action="https://www.paypal.com/cgi-bin/webscr"
method="post">
   <input type="hidden" name="business"
value="amunategui@gmail.com">
   <input type="hidden" name="cmd" value="_donations">
   <input type="hidden" name="item_name" value="Donate to support
these free eBooks :-)">
   <input type="hidden" name="item_number" value="Support">
   <input type="hidden" name="currency_code" value="USD">
   <input type="image" name="submit"
src="https://www.paypalobjects.com/en_US/i/btn/btn_donate_LG.gif"
alt="Donate">
   <img alt="" width="1" height="1"
src="https://www.paypalobjects.com/en_US/i/scr/pixel.gif">
</form>
```

Once you have the code inside your web page, you will see the yellow "**Donate**" button. When you click on it using a valid PayPal account, it will take you to PayPal and ask the donor a series of questions to get the transaction completed. You can customize your message using the "**item_name**" field.

Google Analytics

(No code for this chapter)

Tracking visits on your web property using Flask isn't hard to do but it is still harder than using the free tier of Google Analytics. All one has to do in order to benefit from GA is to create an account, tie a web site to it, and add a few lines of JavaScript on each page you want tracked. Google Analytics will do everything else from that point (and it does a whole lot of analyzing).

Navigate to Google Analytics to create a free account at:

https://analytics.google.com/analytics/web/provision

Once you have signed up, click on blue button at the bottom of the page "**Get Tracking ID**" and accept the terms of service.

Adding the JavaScript Tracker

Under the "**Admin**" tab of the Google Analytics dashboard is the snippet of JavaScript you will need to track your pages. Add your API key where it says:

```
"<<ADD-YOUR-GOOGLE-ANALYTICS-TRACKING-ID>>"
```

and paste this into all the HTML pages you want to track. If you have multiple HTML pages in your Flask project, it makes sense to create a snippet HTML template file just for that and then use an Include code to add it to subsequent pages.

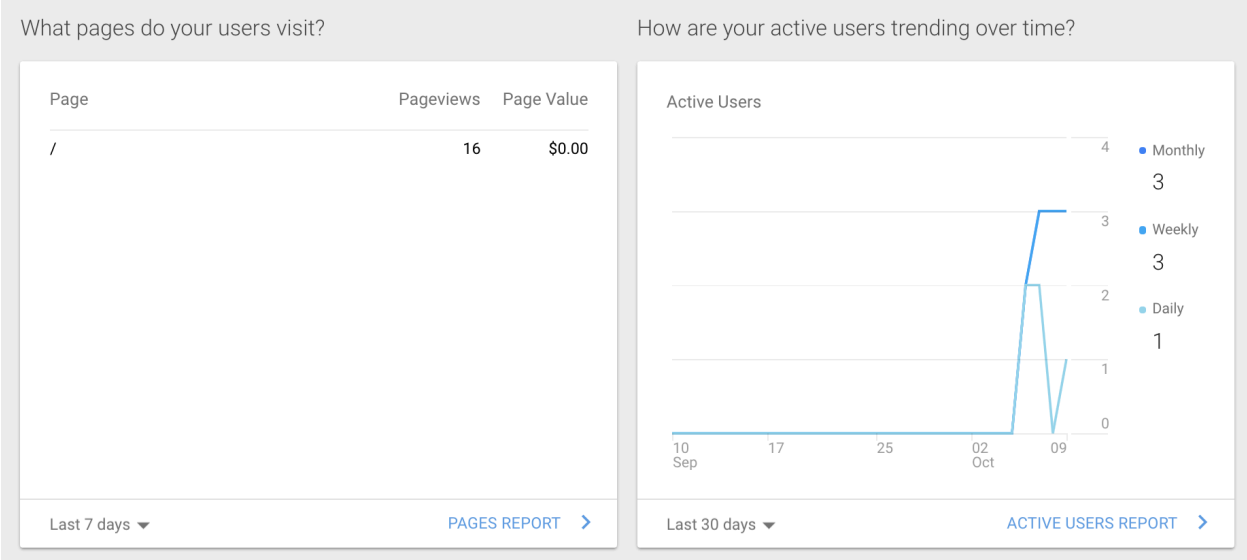
Such as:

```
{% include “google_analytics.html" %}
```

The Google Analytics Report

You could write a book about all the features this tracker offers. In our case, we can keep it simple as we are focusing on quick POCs, the real-time counters, engagements, locations, conversions, etc may not be as useful as the simple counter of visits broken down by pages (keep in mind that it can take days before Google indexes your site, and weeks before you build a following).

Here is a screenshot for a PythonAnywhere experiment I did. As you can see in the left pane, there were 16 pageviews (probably all mine). Either way, once you share the link of your web application to you email list, on social media, etc, this is where you want to look every couple of days to see if traffic is picking up or not.



There is a great class from the Google Analytics training team: https://analytics.google.com/analytics/academy/

Promoting Your Web Application

Let's discuss the various things you should be doing to give your web application the visibility and chance for survival it deserve. This is important stuff, if you can get your cool application in front of enough eyeballs, you application will tank regardless of how cool it is…

Landing Page/Website
Even if you application looks like a web page, you still need a landing site where visitors can get information about you, your company and the cool web applications you are building. That is exactly what I am doing on ViralML.com. The site does house a few web application but it also houses my blog posts, videos, products, thoughts, etc.

Blogging
If you are going to go the organic route, blogging is the single most important things you can do to promote your web application. This isn't something you should wait till your app is out, this is something you should start right this second. It doesn't matter if your application isn't finished, talk about what you are planning to build and how it will help people.

Push out a blog every week and start building a following. This process can be accelerated by sharing your blog posts on other sites like Medium or Reddit (just to name two).

Promoting
Offer people the ability to sign up to your email list and get your newsletter (this can simply be your blog post or a synopsis of the post). Formspree is a great free option to collect emails. Whenever a user signs up, you will get an email with their contact information.

Interacting
This can be done through your blogs and newsletters - ask people questions,

offer users the ability of becoming beta users, etc.

SEO
Make sure it contains the right title tags, the right meta tags, only one H1, and so on and so forth (more on this topic at Steal That SEO Ranking Information and Let Your Content Soar, Guilt-Free)

Mail Campaigns
When you are ready to release, let your users know about it - even give those on your email list the ability of getting early access and solicit reviews and feedback.

Conclusion

You made it to the end! This makes my day! Well, as you can see we can create a full machine-learning driven web application for free. We need to be crafty at times but overall, this works great for quick tests, prototypes and proof-of-concepts. Now, it is up to you to put this techniques to great use - think big!

Also, if you like this eBook, then check out my other education material. These aren't free but will take you much further into the world of ML and web applications (and I appreciate your support):

My Book: Monetizing Machine Learning: Quickly Turn Python ML Ideas into Web Applications on the Serverless Cloud

Take your Python machine learning ideas and create serverless web applications accessible by anyone with an Internet connection. Some of the most popular serverless cloud providers are covered in this book—Amazon, Microsoft, Google, and PythonAnywhere.

You will work through a series of common Python data science problems in an increasing order of complexity. The practical projects presented in this book are simple, clear, and can be used as templates to jump-start many other types of projects. You will learn to create a web application around numerical or categorical predictions, understand the analysis of text, create powerful and interactive presentations, serve restricted access to data, and leverage web plugins to accept credit card payments and donations. You will get your projects into the hands of the world in no time.

My Class: Machine Learning Entrepreneurship - Master Class on Applied Data Science

Become A Machine Learning Entrepreneur! Learn how to transform your ML ideas into fully interactive web applications and paywall subscription sites. Here we will extend multiple Python machine learning ideas into fully interactive web applications, into a format that anybody anywhere can access as long as they have access to a web browser. Our last project will be built around a professional paywall infrastructure so you can control and monetize how and whom can access it.

Best of luck on your data science and entrepreneurial goals!!!

Manuel Amunategui