Machine Learning in Cyber Security

Tutorial - Nov 9th, 2020

Hui-Po Wang

Most of contents created by Shadi and updated by Hui-Po

Agenda

- Setting up a Python Scientific Environment
- Jupyter Notebook
- Scientific Computing (using numpy)
- Data Analysis (using pandas)
- Data Visualization (using matplotlib)

1. Setting up a Python Scientific Environment

- Download Anaconda

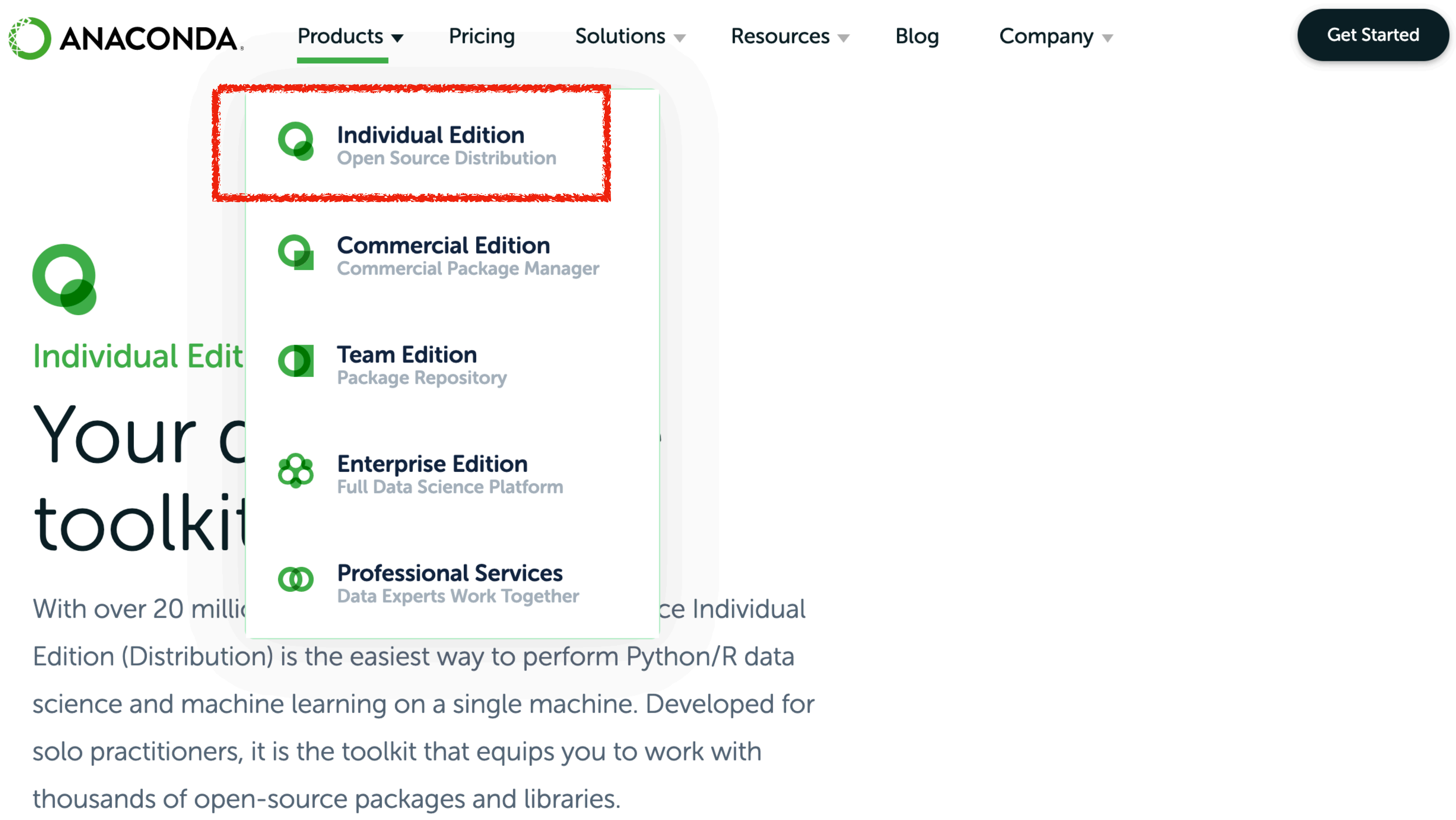
- Install:

```
$ chmod u+x /path/to/Anaconda<...>.sh
$ /path/to/Anaconda<...>.sh
```

- This should already contain most of dependencies required for the course

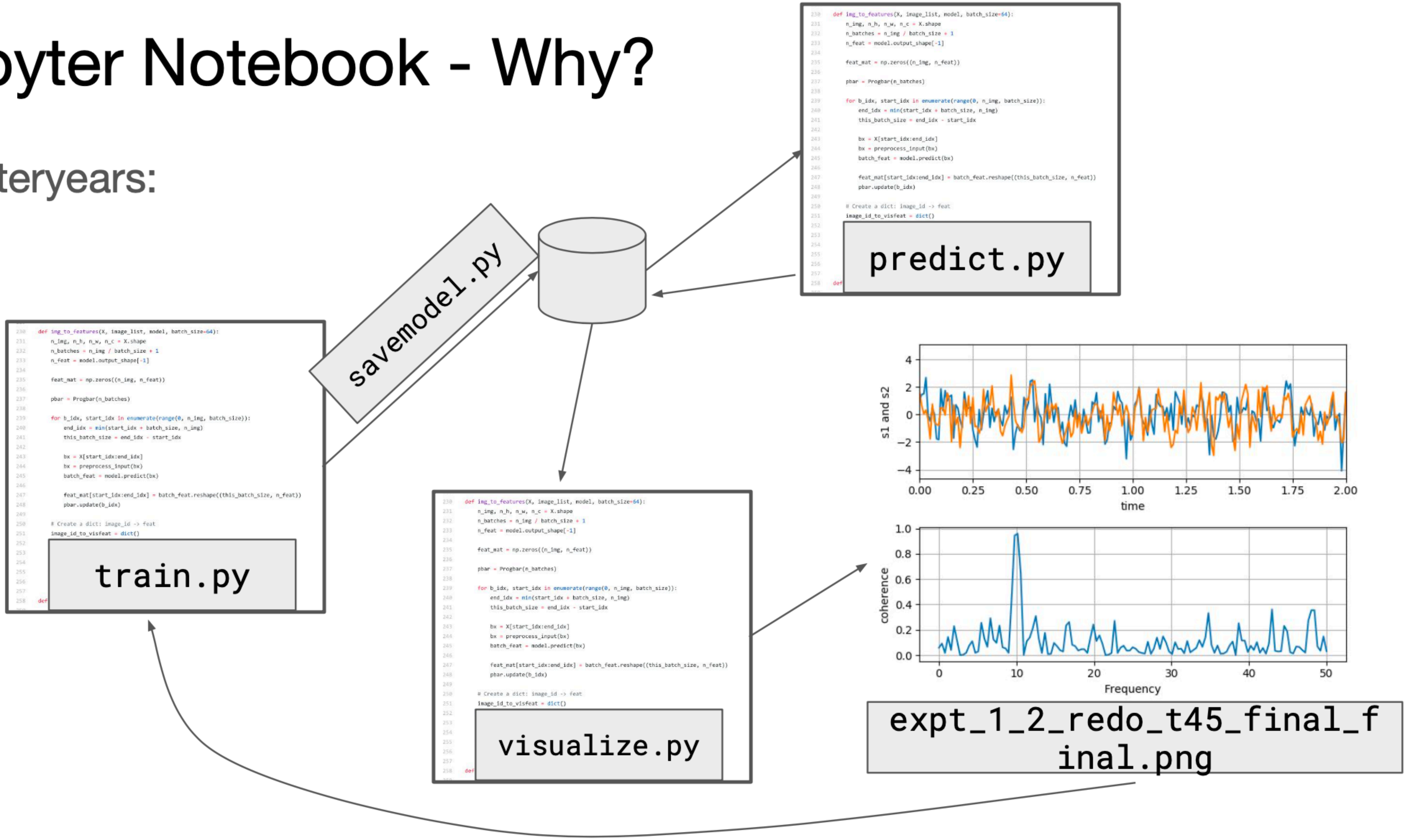
- If not:

```
$ conda install <package-name>  # or
$ pip install <package-name>
```



2. Jupyter Notebook - Why?

In the yesteryears:



2. Jupyter Notebook - Why?

Now: `42.ipynb`

In [36]:
```python
n_epochs = 100
batch_size = 150

with tf.Session() as sess:
    init.run()
    for epoch in range(n_epochs):
        for iteration in range(mnist.train.num_examples // batch_size):
            X_batch, y_batch = mnist.train.next_batch(batch_size)
            X_batch = X_batch.reshape((-1, n_steps, n_inputs))
            sess.run(training_op, feed_dict={X: X_batch, y: y_batch})
        acc_train = accuracy.eval(feed_dict={X: X_batch, y: y_batch})
        acc_test = accuracy.eval(feed_dict={X: X_test, y: y_test})
        print(epoch, "Train accuracy:", acc_train, "Test accuracy:", acc_test)
```

```
0 Train accuracy: 0.93333334 Test accuracy: 0.9311
1 Train accuracy: 0.96666664 Test accuracy: 0.9522
2 Train accuracy: 0.97333336 Test accuracy: 0.9586
3 Train accuracy: 0.96666664 Test accuracy: 0.9607
4 Train accuracy: 0.97333336 Test accuracy: 0.9673
5 Train accuracy: 0.98 Test accuracy: 0.9669
6 Train accuracy: 0.97333336 Test accuracy: 0.9693
7 Train accuracy: 0.96666664 Test accuracy: 0.968
8 Train accuracy: 0.9533333 Test accuracy: 0.9723
9 Train accuracy: 0.97333336 Test accuracy: 0.9683
10 Train accuracy: 0.9866667 Test accuracy: 0.9734
11 Train accuracy: 0.96666664 Test accuracy: 0.969
12 Train accuracy: 0.9866667 Test accuracy: 0.9726
13 Train accuracy: 0.9866667 Test accuracy: 0.9774
14 Train accuracy: 0.98 Test accuracy: 0.9705
15 Train accuracy: 0.9866667 Test accuracy: 0.976
16 Train accuracy: 0.9866667 Test accuracy: 0.9739
17 Train accuracy: 0.9866667 Test accuracy: 0.9709
18 Train accuracy: 0.98 Test accuracy: 0.9736
19 Train accuracy: 0.9866667 Test accuracy: 0.9775
```

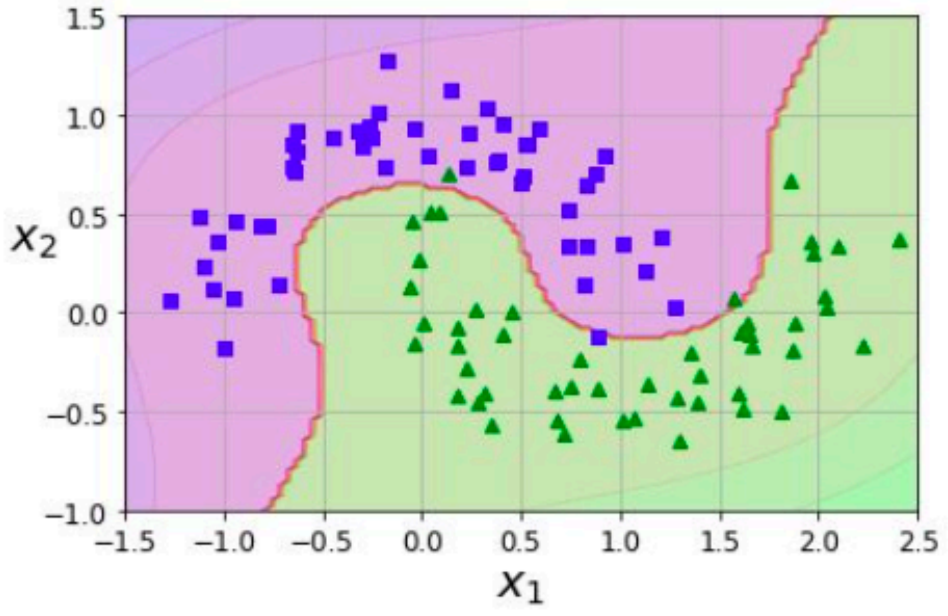
Train models

In [14]:
```python
def plot_predictions(clf, axes):
    x0s = np.linspace(axes[0], axes[1], 100)
    x1s = np.linspace(axes[2], axes[3], 100)
    x0, x1 = np.meshgrid(x0s, x1s)
    X = np.c_[x0.ravel(), x1.ravel()]
    y_pred = clf.predict(X).reshape(x0.shape)
    y_decision = clf.decision_function(X).reshape(x0.shape)
    plt.contourf(x0, x1, y_pred, cmap=plt.cm.brg, alpha=0.2)
    plt.contourf(x0, x1, y_decision, cmap=plt.cm.brg, alpha=0.1)

plot_predictions(polynomial_svm_clf, [-1.5, 2.5, -1, 1.5])
plot_dataset(X, y, [-1.5, 2.5, -1, 1.5])

save_fig("moons_polynomial_svc_plot")
plt.show()
```

```
Saving figure moons_polynomial_svc_plot
```



Predict + Visualize

Computing features matches

- Matching
  - $d_i \in \mathbf{D}_i \rightarrow d_j \in \mathbf{D}_j$
  - Matching is a **nearest neighbor problem**
- In a **low dimensional space**, can be efficiently performed using **KD-Trees**
- However, SIFT descriptors are **128-d** vectors
- Alternatives
  - Approximate Nearest Neighbors (as FLANN library)
  - Brute force matching
  - **Dimensionality reduction**

Consider the two sets of descriptors, $\mathbf{D}_i$ and $\mathbf{D}_j$, computed using a method as SIFT or SURF, for two images I_i and I_j. Matching can be performed using nearest neighbors, just selecting for each $\mathbf{d}_i \in \mathbf{D}_i$ the closest vector $\mathbf{d}_j \in \mathbf{D}_j$. Nearest neighbor queries can be efficiently done representing $\mathbf{D}_i$ in a KD-Tree.

KD-Tree performance is close to brute force for vectors presenting large dimensions. SIFT vectors are 128-d and SURF ones are 64-d. Before matching using KD-Trees, a dimensionality reduction procedure, as PCA, is recommended. Sklearn and OpenCV provide PCA implementations.

In [6]:
```python
from sklearn.decomposition import PCA
pca = PCA(n_components=10)

pca.fit(D_i)
D_i = pca.transform(D_i)
D_j = pca.transform(D_j)
```

- Lowe recomends to compare the **two nearest neighbors**
- In a **good match**, there is a constrast between the two distances
  - Descriptors with no proper match present similar distances between their closest neighbors
- Filtering
  - $d_j \in \mathbf{D}_j$ should be assigned to just **one** $d_i \in \mathbf{D}_i$

Interleave code, plots and explanations (markdown/tex)

Hands-on in Jupyter Notebook

- Create a new environment for the course
  - conda create -n mlcysec python=3.7 numpy matplotlib jupyter nb_conda
  - conda activate mlcysec
- Start Jupyter Notebook
  - jupyter notebook
- Open browser
  - http://localhost:8888

```
    To access the notebook, open this file in a browser:
        file:///Users/wanghuipo/Library/Jupyter/runtime/nbserver-74986-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=d10b546ae3992c2010e05de7e25020431ea8027974373c30
     or http://127.0.0.1:8888/?token=d10b546ae3992c2010e05de7e25020431ea8027974373c30
```