CA400 Testing Document

Ben Kelly - 15337716
Supervisor - Martin Crane

Abstract

SmartPredict is a Web application trading platform which aims to help users make intelligent decisions about buying and selling of currencies (foreign exchange & cryptocurrency markets). This is achieved through machine learning and chart analysis. The application will provide the user with predictions for the market based on the knowledge it has acquired from past data. Recommending whether the user should buy, sell or hold their specific currency according to its current value.A user is able to view the current market sentiment, review the prediction of the machine learning algorithms and interact with charts of past prices data all within the application. The following document outlines the testing methods and strategies utilised to ensure the system is robust, dependable and safe for the end-user. The project can be found online at http://kellyb45.pythonanywhere.com/smartpredict.

Table of Contents

1. Testing Strategy

The testing strategy for SmartPredict has prioritised certain aspects, that is not to say that the other aspects of testing weren't up to standard it just means that more of an emphasis was placed on testing types that are most relevant for the project. The following testing types were completed in order of priority (1 being of the highest priority).

1. User Testing
2. Unit & Coverage Testing
3. AI Testing
4. Environment & Usability Testing
5. Lint testing

When a test failed, the function was examined closely and adjusted according to the criteria it failed on and the test(s) was re-run until it passed. Refactoring was also performed when deemed necessary with any waste and unreachable code removed.

2. Environment Testing

The following browsers and screen sizes were tested. In order to pass the test the following criteria had to be met:

1. The User Interface looked clear and readable.
2. The UI worked as intended, buttons worked, sign in was possible.
3. There were no issues in the console regarding dependent libraries etc.

Note: As SmartPredict is a WebApp it does not support screen sizes of less than 992px in width. The following tests are being run on the online version of the app linked in the abstract. If a user wishes to download a local version they should refer to the User Manual, where a step-by-step guide can be followed to install and run on MacOS and Ubuntu. The app can run locally on other OS's but the syntax for installing and running it will differ slightly.

Browser	**Operating System**	**Screen size small** 992 x 600px	**Screen size medium** 1366px x 768px	**Screen size large** 1920px x 1080px
Chrome	MacOS Mojave 10.14.4	passed	passed	passed
Chrome	OpenSUSE 42.3 64-bit	passed	passed	passed
Chrome	Windows 10	passed	passed	passed
Firefox	MacOS Mojave 10.14.4	passed	passed	passed
Firefox	OpenSUSE 42.3 64-bit	passed	passed	passed
Firefox	Windows 10	passed	passed	passed
Safari	MacOS Mojave 10.14.4	passed	passed	passed
Microsoft Edge	Windows 10	passed	passed	passed

3. Unit Testing

Comprehensively unit testing elements of the app which were more susceptible to containing bugs (such as the views) was extremely important to the overall welfare of the project. The full list of unit tests can be found at `smartpredict/tests.py`

3.1 Registration & Login

It was important to unit test all pages to ensure they were viewable and contained the right information given the user was logged in.

```
class LoginFunc(TestCase):

    def test_login(self):
        c = Client()
        response = c.post('/smartpredict/login/', {'username': 'john', 'password': 'smith_12'})
        self.assertEqual(response.status_code, 200)
        self.assertRedirects(response, '/smartpredict/')
        self.assertContains(response, 'john')
```

3.2 Dashboard

The Dashboard was tested by ensuring it returned the correct response, used the appropriate URL, contained the correct information and used the correct template. This process was repeated for all the major views (discussed on page 5, Coverage).

```
class DashBoardTest(TestCase):
    def setUp(self):
        self.client = TestClient()
        user = User(username="john")
        user.save()
        self.client.login_user(user)

    def test_dashboard_status_code(self):
        response = self.client.get('/smartpredict/dashboard/')
        self.assertEquals(response.status_code, 200)

    def test_view_url_by_name(self):
        response = self.client.get(reverse('dashboard'))
        self.assertEquals(response.status_code, 200)

    def test_view_uses_correct_template(self):
        response = self.client.get(reverse('dashboard'))
        self.assertEquals(response.status_code, 200)
        self.assertTemplateUsed(response, 'smartpredict/dashboard.html')

    def test_dashboard_contains_correct_html(self):
        response = self.client.get('/smartpredict/dashboard/')
        self.assertContains(response, 'Dashboard')

    def test_dashboard_does_not_contain_incorrect_html(self):
        response = self.client.get('/smartpredict/dashboard/')
        self.assertNotContains(response, 'Hi there! I should not be on the page.')
```

3.3 Forgotten password email client

When a user forgets their password, they can go to the forgotten password page, in which an email will be sent to reset it for them. This piece of code confirms that the server is sending the email. Examples of these emails can be seen in `ben/sent_emails`

```
class EmailTest(TestCase):
   def test_send_email(self):
       # Send message.
       mail.send_mail(
           'Subject here', 'Here is the message.',
           'from@example.com', ['to@example.com'],
           fail_silently=False,
       )

       # Test that one message has been sent.
       self.assertEqual(len(mail.outbox), 1)

       # Verify that the subject of the first message is correct.
       self.assertEqual(mail.outbox[0].subject, 'Subject here')
```

3.4 AI & Sentiment

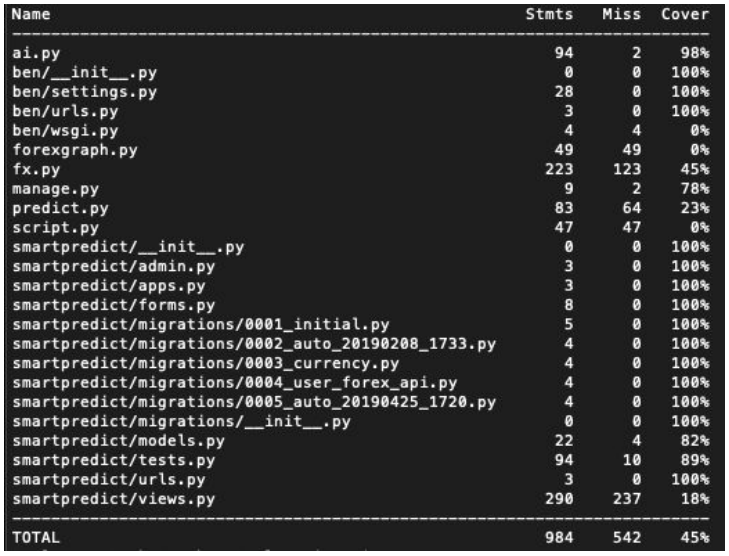
The AI is further tested in section 7, these tests are simply ensuring the basic functionality is correct. The sentiment analysis page is not directly linked anywhere in the application but is used within the respective dashboards, it is important that this piece of functionality is tested as it is used by numerous views.

```
class TestAI(TestCase):
   def test_fx(self):
       predict = main()
       self.assertIsNotNone(predict)

   def test_crypto(self):
       predict = maincry()
       self.assertIsNotNone(predict)

class SentimentViewTests(TestCase):
   def test_sentiment_view(self):
       w = self.create_sentiment()
       url = reverse('smartpredict/sentiment.html')
       resp = self.client.get(url)

       self.assertEqual(resp.status_code, 200)
       self.assertIn(w.title, resp.content)
```

4. Coverage

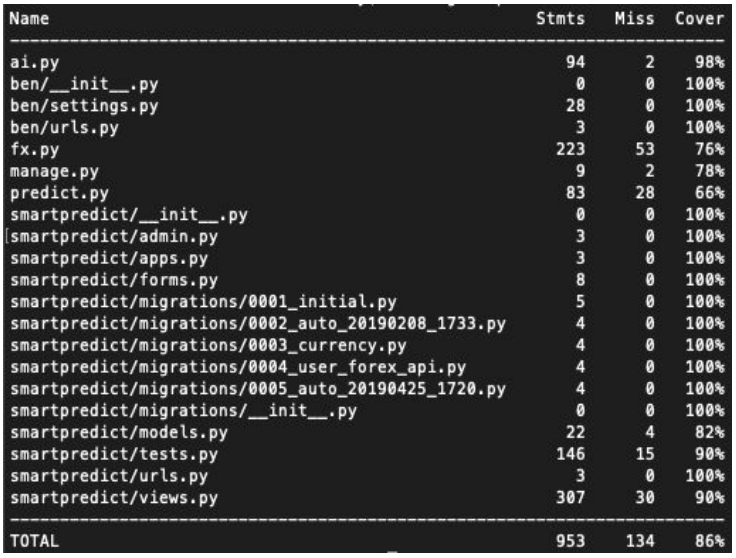
When writing the unit tests I wanted to make sure everything was tested. I checked this by using Coverage.py this gave me a breakdown of what I was catching and missing in terms of statement execution. This helped significantly as shown below. After I had finished writing the code for all of my unit tests I ran coverage to discover I hadn't even tested the majority of the app (45% total coverage). After some refactoring and re-testing, I managed to get this figure up to 86%.

Before:



Name	Stmts	Miss	Cover
ai.py	94	2	98%
ben/__init__.py	0	0	100%
ben/settings.py	28	0	100%
ben/urls.py	3	0	100%
ben/wsgi.py	4	4	0%
forexgraph.py	49	49	0%
fx.py	223	123	45%
manage.py	9	2	78%
predict.py	83	64	23%
script.py	47	47	0%
smartpredict/__init__.py	0	0	100%
smartpredict/admin.py	3	0	100%
smartpredict/apps.py	3	0	100%
smartpredict/forms.py	8	0	100%
smartpredict/migrations/0001_initial.py	5	0	100%
smartpredict/migrations/0002_auto_20190208_1733.py	4	0	100%
smartpredict/migrations/0003_currency.py	4	0	100%
smartpredict/migrations/0004_user_forex_api.py	4	0	100%
smartpredict/migrations/0005_auto_20190425_1720.py	4	0	100%
smartpredict/migrations/__init__.py	0	0	100%
smartpredict/models.py	22	4	82%
smartpredict/tests.py	94	10	89%
smartpredict/urls.py	3	0	100%
smartpredict/views.py	290	237	18%
TOTAL	984	542	45%

After:



Name	Stmts	Miss	Cover
ai.py	94	2	98%
ben/__init__.py	0	0	100%
ben/settings.py	28	0	100%
ben/urls.py	3	0	100%
fx.py	223	53	76%
manage.py	9	2	78%
predict.py	83	28	66%
smartpredict/__init__.py	0	0	100%
smartpredict/admin.py	3	0	100%
smartpredict/apps.py	3	0	100%
smartpredict/forms.py	8	0	100%
smartpredict/migrations/0001_initial.py	5	0	100%
smartpredict/migrations/0002_auto_20190208_1733.py	4	0	100%
smartpredict/migrations/0003_currency.py	4	0	100%
smartpredict/migrations/0004_user_forex_api.py	4	0	100%
smartpredict/migrations/0005_auto_20190425_1720.py	4	0	100%
smartpredict/migrations/__init__.py	0	0	100%
smartpredict/models.py	22	4	82%
smartpredict/tests.py	146	15	90%
smartpredict/urls.py	3	0	100%
smartpredict/views.py	307	30	90%
TOTAL	953	134	86%

As a result of these extra tests being run, some underlying bugs were unearthed, waste was removed and any unreachable code was either removed or refactored.

5. Lint Testing

PyLint was utilised to uncover any inherent errors in the source code, this could stem from bad naming conventions, bad indentation (which is particularly important in python), unused arguments and it implements an overall coding standard which is helpful. The lint testing was run, which identified some inconsistencies, these were amended and the final scores for the most used python files can be seen below.

manage.py	
smartpredict/views.py	
ai.py	
fx.py	

This testing made the code much more maintainable and readable.

6. User Testing

The user testing questionnaire can be found at in the GitLab repository at
https://gitlab.computing.dcu.ie/kellyb45/2019-ca400-kellyb45/blob/master/docs/documentation/User_Testing.pdf

The aim of the user testing was two things:

1. To ensure the User Interface was usable, intuitive and clear (qualitatively)
2. To measure how usable and intuitive it was (quantitatively)

To achieve aim 1, I first asked how proficient the user was at using technology and how often (if at all) they trade. These two things would give a huge advantage when asked to perform tasks on the application later. I then asked some general questions about the overall design of the app to try gage a consensus of how the app was received.

To achieve aim 2, the user was asked to perform a set of tasks (e.g. log in and navigate to the dashboard) while being timed. The times it took the participants to complete a set of tasks were then compared to the time it takes an expert in the app (the creator) to achieve the same task. If the times were within 0-5 seconds they were deemed as a success, outside of this mark the user had either moderately failed (<10 seconds outside of time window) or badly failed (>10 seconds).

Ten users were tested across a wide demographic (ranged from 21-year-old computing student with some trading experience to an 87-year-old man who rarely uses the web and zero trading experience).

6.1 Registration & Login

- Create an account and use it to log in

All users found this task fairly menial, with some users initially entering a password that wasn't valid, this was quickly amended after feedback from the system.

6.2 Dashboard Activities

- Navigate to your Cryptocurrency dashboard
- Check
    - Current Bitcoin price
    - Current Ethereum sentiment
    - Expected closing Bitcoin price
- Navigate to your Forex Dashboard and check RSI of EUR/USD

All users could easily navigate to the general Dashboard with some having difficulty deciding on which market to select for the given currency. The users with a lack of knowledge pertaining to trading struggled slightly with determining what the sentiment of a specific currency was but the 7/10 managed to pass this task which was deemed a success. Some users didn't initially realise the charts were interactive and didn't utilise this functionality but this aspect can be viewed as a bonus and not a critical piece of functionality.

6.3 About page & logout

- View the functional specification
- Go to help section
- Logout

These tasks were again found trivial by eight of the ten participants with the other two graded as a moderate failure.

Overall the user testing was a huge success, It proved that the application has very high usability particularly when used by SmartPredict's 'target audience' but certainly usable outside of that with a small learning curve. Some improvements made to the app as a result of the user testing were.

- Instead of downloading the functional spec when navigating to help section, it was opened a new tab with the pdf within it.
- The footer overlapping with some graphs at the end of both dashboards.

7. AI testing

It is important to note that while developing and testing both algorithms I made a conscious effort to keep 'tinkering' to a minimum, This would cause overfitting of the small dataset I had of 2000 data entries for Bitcoin and ~ 4000 data entries for each Forex currency pair.

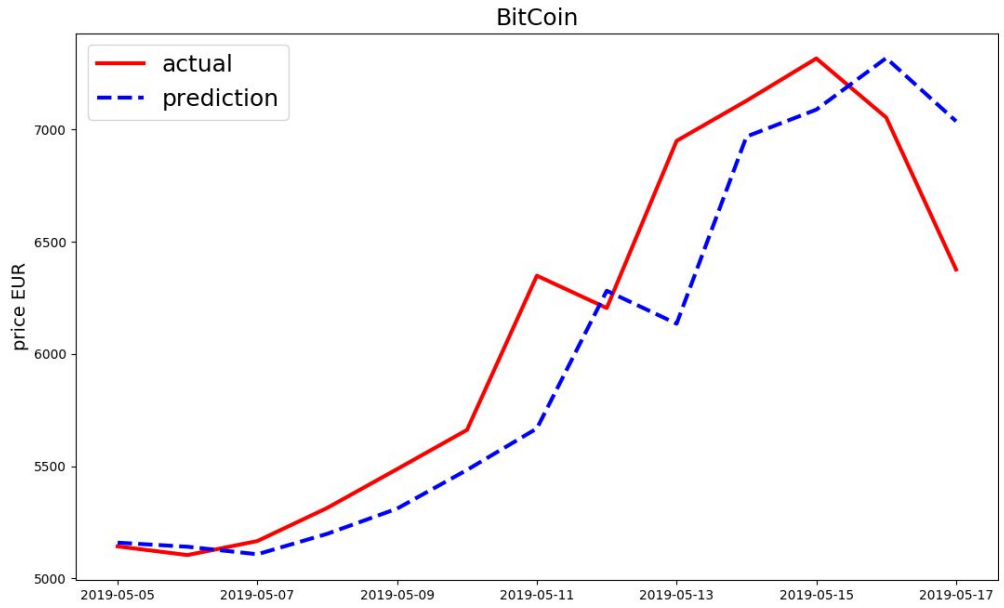
7.1 Cryptocurrency AI

While testing both AI's a deliberate attempt was made to prevent overfitting, while this was less of a problem with the crypto LSTM model due to the volatility of Bitcoin an effort was made to only re-run the function when absolutely necessary. To minimise overfitting a dropout value of 0.25 was used to set random nodes in the NN to zero and normalisation was also utilised. To minimise error a combination of two main factors were experimented with. They were also tested using two loss functions, mean absolute error (MAE) and mean squared error (MSE).

- No. of neurons
- No. of epochs

```
Error:0.02540510253273065 lstm_neurons:20 epochs:20 loss:mse
Error:0.022855874390560266 lstm_neurons:20 epochs:50 loss:mse
Error:0.023686172909045726 lstm_neurons:30 epochs:25 loss:mse
Error:0.022373680197217676 lstm_neurons:20 epochs:20 loss:mae
Error:0.021694757455838164 lstm_neurons:20 epochs:100 loss:mae
Error:0.02193876019376793 lstm_neurons:30 epochs:50 loss:mae
Error:0.022392384178562638 lstm_neurons:30 epochs:25 loss:mae
Error:0.021512669211953196 lstm_neurons:20 epochs:30 loss:mae
```

As observed there is no obvious performance enhancement observed by increasing the number of neurons or epochs past 20 and 30 respectively. For efficiency, we will take these values going forward. As shown below the algorithm itself adjusts a day late with the least difference in daily prices working best, this is unfortunate but expected as Bitcoin is so volatile.



7.2 Forex AI

As the foreign exchange market is much more robust against change compared to the crypto market the MSE and MAE values were expected to be much lower. The model uses ARMIA (AutoRegressive Integrated Moving Average) instead of LSTM as this seems to be more accurate for Forex prices in the documentation I've read. The model takes daily price points from 2005 to the present day and uses a 90:10 split for training: testing (the same split as the Crypto AI).

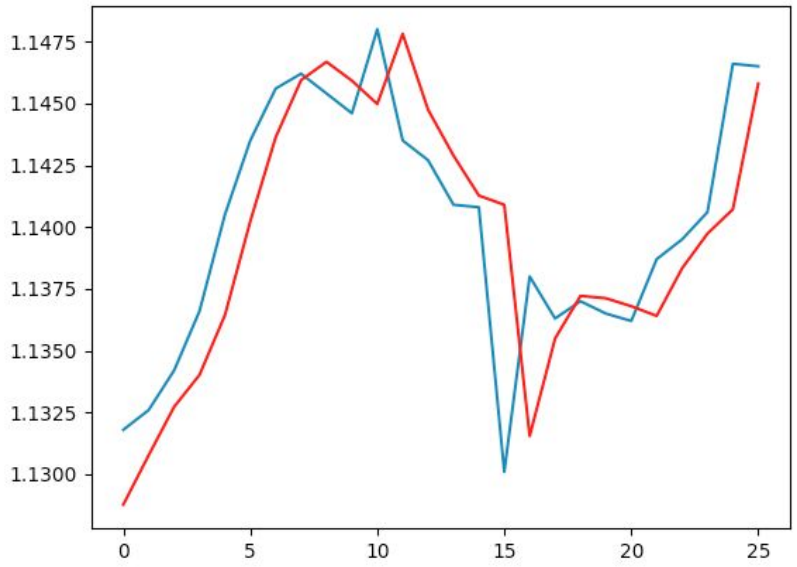
In general, the algorithm has good accuracy with the forex algorithm ranges ~ (0.002-0.004) mean absolute error (MAE) depending on the currency pair. The difference in accuracy is expected because of the highly volatile nature of cryptocurrencies. This relatively high error in Bitcoin is perhaps less risky than a smaller percentage of inaccuracy for forex and the margins for profit are much smaller.

```
Actual=0.864200, Predicted=0.864299
Actual=0.862900, Predicted=0.864168
Actual=0.865000, Predicted=0.862875
Actual=0.866300, Predicted=0.865125
Actual=0.864900, Predicted=0.866342
Actual=0.865200, Predicted=0.864832
Actual=0.865200, Predicted=0.865264
Actual=0.867300, Predicted=0.865277
Actual=0.867400, Predicted=0.867378
Actual=0.864600, Predicted=0.867388
Test Mean Absolute Error: 0.002
```

Three currency pairs are supported by the project, with both sides of EUR/GBP for particular interest due to Brexit. The full list is:

- EUR/USD
- EUR/GBP
- GBP/USD
- GBP/EUR

Here is a graph of EUR/USD with the blue line being the actual data and red being the algorithm's prediction of the final 25 day period.



8. Conclusions

Overall, the testing of the project has been as comprehensive as possible, they say you should never be surprised with testing but I was encouraged by the results. The unit and coverage tests are a testament to the overall breadth of our search with the AI. With the user testing a good example of how in depth the testing was. Unfortunately, both dashboards are limited in terms of refresh rate, due to the price data relying on external APIs the project is limited to 3 calls per minute if this is exceeded the app will remain up and running but the data will not be displayed. I am pleased with how positively the project has been received and feel like the app can be of genuine use to someone who is interested in the field. I am confident in saying the app is robust enough for its use cases as a result of the comprehensive testing documented here. The risk of trusting the AI can be slightly mitigated due to its accuracy exhibited, however, it should be noted it is impossible to predict a volatile currency such as Bitcoin all the time.

The most valuable form of testing was the user testing, real-life feedback can only improve the app and its usability. Little tweaks make a huge difference to users when it comes to the daily use of the application.