TF2MarketAnalyst

A Team Fortress 2 Trading Project

Introduction

Bots are a big part of modern TF2 trading. This is easily seen by any Backpack.TF classifieds page, the sheer number of automated traders usually outweighs the human ones by a large amount. Gladiator.TF bots are especially prevalent, and I would guess are the most competitive by quite some distance as they are capable of automatically adjusting their pricing.

Gladiator.TF is not a cheap service to use, especially when it comes to automatic pricing bots, costing around $20 a month to run. The biggest reason behind this, I'd also guess, is due to the sheer quality and quantity of the data that these bots are capable of leveraging. As far as I'm aware there's quite some overlap between the teams behind Backpack and Gladiator, so an awful lot of private data is giving them the edge.

Despite all of this, Backpack.TF still offers a wealth of useful data through their API. And Gladiator bots don't exactly rule the market with an iron fist. As will be shown in this document, there is a lot of space in the market for competition, and it's my goal to leverage that to create a competitor automatic pricing bot, with very low overheads, offered to the community at large completely for free.

This document will outline the process behind that, the preliminary and full analysis as well as the development of the fully automated bot suite. I would also like to persistently update this with the performance of my own bot, once deployed.

Motivation

- Money
- Persistent interest in the TF2 virtual economy.
- Sweet, sweet moolah
- Developing my analytics and software engineering skills.
- ~~£££££££££££££££££££~~

Objectives

- Identify pricing gaps and profit opportunities in the market.
    - Build datasets using the Backpack.tf API.
    - Investigate the classifieds listings available.

- Use a bot script to leverage these gaps effectively and automatically for profit.
    - Create an initial pricelist from the data provided in the analysis phase.
    - Build the necessary scripts to update the pricelist for maximum effectiveness.
    - Build a functional Steam Trade bot.

- Monitor bot progress.

Analysis

Initial Observations

For the uninitiated, Backpack.tf is a website that stores statistics about the Team Fortress 2 virtual economy. While best known for their comprehensive lists of price data, they also host classifieds listings for TF2 items. These take the form of sell orders (users who have an item that are looking to sell it at a given price) and buy orders (don't have, looking to buy).

A look at a page for any given item on the site will yield some number of these classifieds. A standard classifieds page will look much like figure 1, with a visible price gap between the lowest sell order and the highest buy order. In the case of figure 1, this gap is significant (for low-tier trading). The gap being about .5 keys, there's space for a profit seeking-trader to jump in that gap.

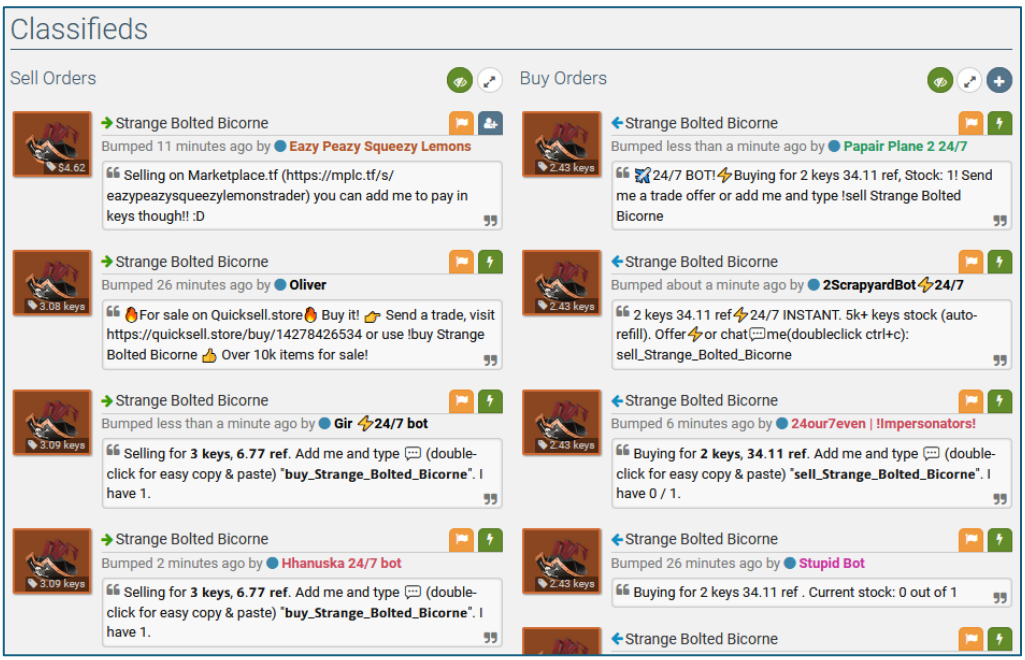


Figure 1

Once upon a time this was the case, but this is no longer true in modern TF2 trading, for several reasons. Namely:

- Low trade volume means that buy orders could sit for months. In the time it might take for the buy order to be fulfilled, the lowest selling price could have dropped.
- Capped classified listings mean that manual (non-premium) traders can only have 50 buy/sell orders active at any given time.
- Prevalence of bots in the marketplace means that impatient quick sellers (that are likely to look to buy orders instead of listing their own sell order) are very, very unlikely to ever choose a manual trader, even if their offer is higher than a bot's.
- Competitive auto-pricing bots mean that unless the manual trader spends lots of time monitoring their buy orders, they are most likely to be outbid by said bots. Even if they do monitor the listings, the overall profit to be made decreases every time the bots update their prices.

Now, while a dumb, manual price trading bot can mitigate some of these issues, it can only do so to a very limited extent. It's clear from a glance at the majority of these classifieds' pages that auto-pricing bots have a gigantic edge on the competition.

It's also true that you're always competing with Scrap.TF, which once again uses many automatic price-adjusting bots.

Backpack.TF API

Backpack.TF provides an API to retrieve pricing data. To begin my analysis, I grabbed the existing price sheet that they have made available. This does not cover all items in the game, but it does have data for 4989 of them. This had to be adapted into a usable form for classifieds searching, and due to a very confusing schema within the JSON I was provided with, meant that items of unusual quality had to be excluded.

> *To digress, the Backpack.TF prices are curated by its users. A price can be suggested for each item and is then voted on by other users until a verdict is reached. With interest in the game and its economy waning in recent years, a lot of this price data is extremely outdated. So, while the actual prices in the price sheet were inaccurate, they were still useful enough (for the most part) as a guideline.*

The next step was to get the classifieds data for these items. This could be gathered by grabbing the item name and quality from the price sheet and converting them to SKUs (Stock Keeping Unit names). Getting the classifieds data was more challenging, with rate-limiting on the Backpack.TF API. The current version of the script used to get this data has an interval of 1 second between classifieds snapshot requests.

Deciphering the classifieds snapshot data was the next issue. An older version of the API would provide all the classifieds' data for a given item, but it has been deprecated in recent years. The new API provides a 'snapshot' instead, with a relevant subset of the overall data. From these snapshots the lowest sell price and the highest buy price can be extracted, building the basis for our automatic pricing, and revealing some interesting insights.

I used the Python API wrapper to gather all the data, which is sadly out of date and bits of it needed rewriting (specifically related to classifieds snapshots). (I did this inside of my own script, but if I get the chance, I will fully rewrite the BP.TF wrapper).

Further Investigation

This script (available on my GitHub!) would return a CSV with columns featuring the item's SKU, the gap between the buy and sell prices (sell – buy), and my own metric for approximating trade volume (for more information about this, and exactly why it's important, check out appendix 1).

The first few versions of this yielded some promising results, mixed in with some very confusing ones (figure 2.1).

Strange Fed-Fightin' Fedora		38.84	4
Strange HazMat Headcase		-1442	12
Strange Crone's Dome		-2628.78	23
Haunted Crone's Dome		-4607.78	30
Strange Barnstormer	2.780000000000001		1
Strange Brotherhood of Arms		-308.62	29
Strange Well-Rounded Rifleman		-15	9
Strange Pom-Pommed Provocateur	11.780000000000001		8
Strange Counterfeit Billycock		-1058.333333	2
Strange Wraith Wrap	1.8900000000000148		8
Strange Antlers		-3432	16
Strange Bonk Leadwear		-13.45	14
Strange Bolted Bicorne	51.55000000000001		5
Strange Firewall Helmet	1.4399999999999977		4
Strange Texas Tin-Gallon	14.840000000000003		10
Strange Bunsen Brave		19.11	6
Strange Burning Bandana	4.780000000000001		11
Strange Katyusha	30.769999999999996		4
Strange Das Naggenvatcher	39.879999999999995		2
Strange Valley Forge	0.21999999999999886		8

Figure 2.1

An abundance of negative results in the gap field (some huge – 1000 ref is roughly equivalent to 14 keys at the time of writing) in the dataset raised questions. The reality of this is that Backpack.TF isn't built to handle the abuse it receives these days. Traders – in their infinite wisdom – covet limited items, and spelled items are as limited as they get. There's no filter for spells - they end up with all the other classifieds, so bot traders create the most bizarrely priced buy orders for cheap items to snap up the spelled ones (figure 2.2).

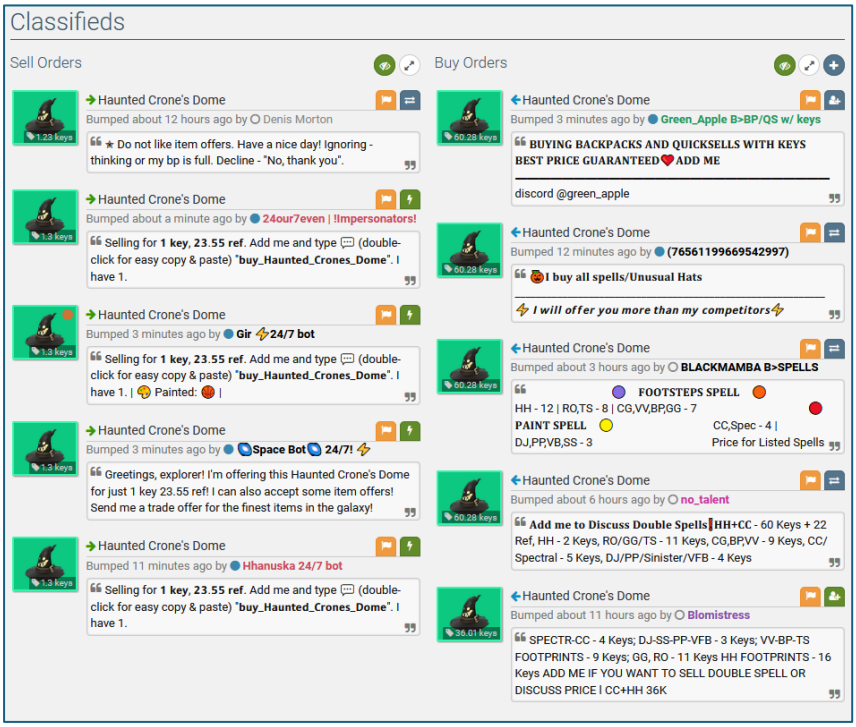


Figure 2.2

Overcoming this involved using some very basic statistical techniques and a simple assumption. It's very unlikely that the highest real buy order is any more than 110% of the lowest sell order, so if you set that as a constraint then you get real values for those buy orders.

This doesn't stop the actual classifieds from being flooded with these nonsense orders – but it does give a better idea of what the market really looks like from a human trader's perspective.

A more complete, filtered version of a similar dataset (with a full version of the volume metric implemented) gave a clearer view. An easy assumption to make is that items with a higher volume have more competition, and therefore lower margins. The data collected proved this to be true in most cases, but to make sure and check for outliers, I graphed it.

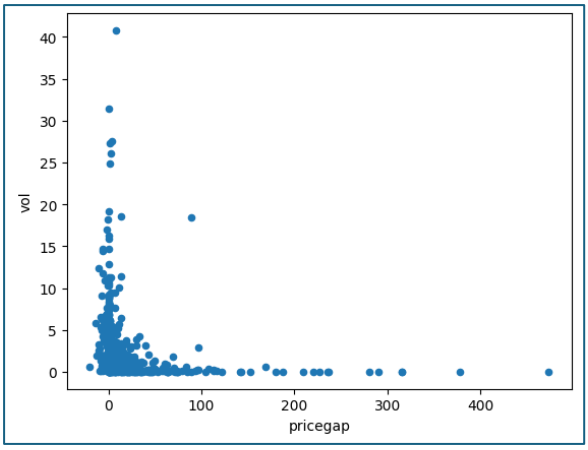


Figure 3.1: Vol against Price Gap for items priced 1-2 keys

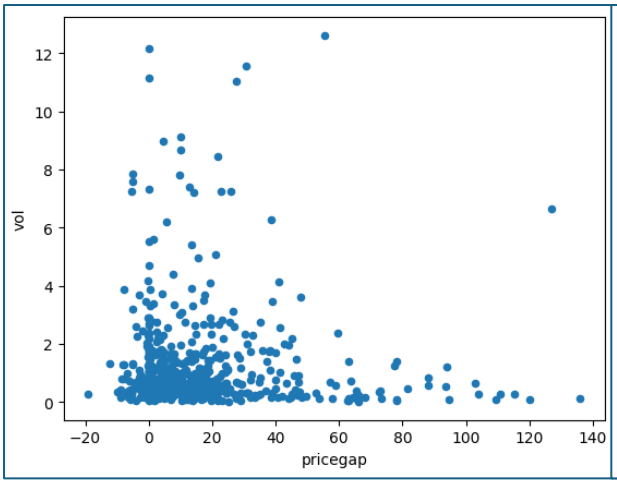
As cluttered as this visualization (figure 3.1) is, it does say a lot about the current state of the market. Most items are low volume and low margin. The few that have a high volume have low margins, and the few that have high margins are low volume. Initially, the interesting parts of this graph are the outliers.

The most obvious outlier is an item with a volume metric of close to 20, and a price gap of nearly 100. This item happens to be the Strange Lucky Shot, a soldier cosmetic item. The reason for such a high volume is the sheer number of sell listings, and the price gap is due to a lack of buy listings. It looks that an automated bot has priced the buy at 0.05 refined because there are no other buyers looking for a non-spelled and non-parted version of the item.

If I had to guess, the cosmetic is not particularly popular, and most trade bots are overstocked on it, which caused the lack of buy orders, causing the large price gap.

Figure 3.2

This does reveal an obvious flaw in the way that the approximate volume is calculated, so that had to be revised, detailed again in appendix 1. Revising the volume metric gave a more interesting and varied visualization, with results that made more sense.



	sku	pricegap	vol
139	Strange Harmburg	0.11	12.177
142	Non-Craftable Brainiac Hairpiece	55.33	12.614
236	Taunt: The Fist Bump	5.44	6.210
279	Taunt: The Crypt Creeper	0.11	11.132
323	Non-Craftable Pink as Hell	10.00	9.119
334	Non-Craftable Robo Community Crate Key	22.89	7.249
362	Aqua Summer 2013 Cooler Key	38.67	6.277
365	Brown Summer 2013 Cooler Key	30.56	11.564
378	Festive Sapper	1.44	5.612
408	Strange Part: Kills with a Taunt Attack	0.11	7.329
494	Harlequin's Hooves	21.78	8.438
498	Strange Cop Caller	127.11	6.649
516	Batter's Bracers	27.55	11.042

Figure 3.3: Vol against price gap, alongside high-volume items

The high-volume items here make a lot more sense, the list is mostly populated by keys, strange parts, paints, and taunts. These items – for the most part – can be used by all classes in the game and as such are some of the most traded. Occasionally, other items slip into this list, but this is still the result of conveniently timed listings.

The next and effectively final stage in the analysis process is to automate the process of finding the right items for the bot to create buy listings for based on these two metrics. The idea is to create a single "fitness" value for every item in the dataset, rank based on that value, and then have the bot create listings for the highest-ranked items.

After some brief experimentation, the most promising formula for a fitness value was:

$$fitness = \log_2 pricegap * vol$$

This prioritized the vol value over the price gap but ensured that items with a high enough price gap would still receive a good fitness value.

Other Insights

[WIP]

Implementation

External Resources and Dependencies

To run this project the following Python libraries are required:

- NumPy (1.26.2)
- Pandas (2.0.3)
- Requests (2.28.1)
- Urllib3 (1.26.13)
- BackpackTF (0.1.4) - https://pypi.org/project/BackpackTF/

For the next stage, the following resources are required:

- tf2autobot (5.12.2) - https://github.com/TF2Autobot/tf2autobot/

I chose to deploy my version of the bot (both for development and production) on an AWS EC2 instance.

Development

Since tf2autobot is so versatile, the trading algorithm (all the associated analytics scripts and the fitness scoring system) became a plugin for it.

Final Implementation

Evaluation

Performance

Next steps

Installation and Operating Instructions

Using the Python scripts

If you're just looking to use the Python scripts to perform your own analysis, it's very simple. This involves populating the masterlist.csv price sheet, then visualizations can easily be created (a notebook for this is packaged, in "Price data vis.ipynb".

To populate the price sheet, or use any Backpack.TF functionality, a private key and token are required. These need to be placed in a file named "keys.txt", with the API key on the first line, and the User token on the second line.

More info on how to get these can be found here:
https://github.com/TF2Autobot/tf2autobot/wiki/Before-you-start

(OPTIONAL) Cloud Deployment

I chose to deploy my version of the bot (both for development and production) on an AWS EC2 instance.

Setting this instance up is trivial, and guides on it are easy to find, so I'll skip over that and cover how to install tf2autobot on Amazon Linux. For the bot to function, all the dependencies must be installed, namely Python3 and Nodejs. To do this, install the packages by running:

```
Yum install python3.11

Yum install Nodejs
```

Both come with their respective package managers (pip for Python and npm for Nodejs), so those can be used to resolve the rest of the dependencies.

The next step is to install Git:

```
Yum install git
```

From here, follow the guide on setting up tf2autobot on Linux:

https://github.com/TF2Autobot/tf2autobot/wiki

Using the bot scripts

Populating the price sheet

To populate the price sheet, or use any Backpack.TF functionality, an API key and token are required.

Setting up auto-refresh

Configuring the Steam bot

Appendix 1: The Market and You(r bot) – A guide to Trade Volume

Introduction

I'll keep this brief – bots exist to make money – more specifically profit. Bots have overheads, so an unprofitable bot is better off dead. To keep your bots alive, you need to create profit fast enough.

The easiest way to mitigate this effectively is to understand volume and its impact on your trades. Items that are traded often have high volume, their classifieds get lots of traffic, they are not only desirable but exist in such a quantity that they are traded a lot. Ideal for a market profiteer to deal in.

Volume too high causes problems too. Mann Co. Supply Crate Key trading is high volume, sure. But it's very low margin and ***extremely*** unpredictable. To a degree that algorithmic trading is very, very risky.

Once again – I'm a speculator, with limited evidence and no formal background in finance or economics.

Volume, volume, and not a dB to hear.

Any competent TF2 bot trader cares about two metrics: margin and volume. Margin is the money you make, and volume is how likely – and how fast – you make it. Calculating your margins is easy, volume is harder. Backpack.TF data is very limited in scope. You can't see what's happening in the actual trades like a real market. Only the number of classifieds listings currently active.

With this information, buy orders are functionally useless. Bots with unlimited classified listings populate the buy order section of virtually every item in the database. So only sell listings are helpful. A very primitive volume metric might just be this – the number of sell listings. However, to make this more useful, time must be accounted for.

So, approximate volume *could* be derived from seeing how long these sell listings live for. Old sell listings are a sign of low volume, newer ones signal a higher volume. There are smart ways to calculate this based on historical listing data (that I haven't quite figured out yet), but the easiest way to figure this out quickly is:

$$approx\ vol.metric = \frac{sell\ listing\ count^2}{avg(listing\ relist\ timestamp - \ creation\ timestamp)}$$

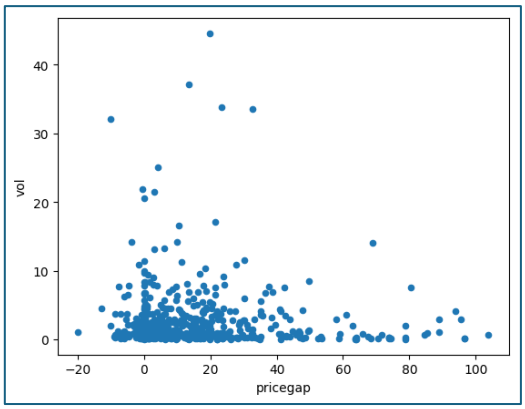
This, when used across the average of all sell listing timestamps, should give a vague idea on how long the sell listings tend to live for; hence we have volume.

Lucky Shot?

The approximate volume statistic revealed an interesting edge case, where an overstocked undesirable item ended up having a high value for volume. This was due to the unusual weighting of the number of sell listings in the calculation. The reason this was done is specified earlier. Any imbalance between the number of buy/sell listings is probably a bad sign. A way to mitigate this is to alter the earlier formula. The denominator must be multiplied by absolute value of difference between the listing counts.

$$approx\ vol.metric = \frac{1}{(1 + abs(buy - sell)) * avg(age)}$$

This was then tested and graphed to display any variation in the dataset. Given the trends observed so far, the most likely result is an asymptotic graph.



While not perfect, the asymptotic pattern is mostly preserved. A deeper dive into the items with the highest vol values gives more insight.

	sku	pricegap	vol
45	Genuine Arkham Cowl	19.67	44.558
96	Strange Pencil Pusher	4.22	25.000
106	Strange HazMat Headcase	-0.56	21.949
138	Strange Cloud Crasher	23.45	33.857
326	Non-Craftable An Extraordinary Abundance of Tinge	3.00	21.479
368	Green Summer 2013 Cooler Key	21.34	17.071
371	Non-Craftable Blue Summer 2013 Cooler Key	10.56	16.604
417	Strange Part: Kills with a Taunt Attack	0.11	20.608
497	Non-Craftable End of the Line Key	13.44	37.167
528	Batter's Bracers	32.61	33.571
640	The Last Laugh	-10.00	32.055

Most of the items with high vol stats are keys, paints, and parts, which is to be expected. The big issue now is that items that have been listed recently massively skew the vol stats. This needs to be addressed before deployment. There are 2 possible ways to handle this:

- Exclude new listings
- Leverage historical data

I chose to exclude listings created in the last 30 seconds, opting to include their prices in the dataset but setting their timestamps back an hour.