CS Capstone Progress Report

March 20, 2019

Auction Hunter

Prepared for

Ryan Kalb

Prepared by

Group 4
Auction Hunter
Alexander Hull
Alexander Jacobson
Yufei Zeng

Abstract

Progress report which contains our teams successes, challenges, and work still needs to be completed. This encompasses the work completed during Winter 2019 term.

CONTENTS

1 Introduction

Our Auction Hunter project is meant to make searching for salvaged car auctions much easier for the user. Auction Hunter aims to help users get a better deal and know what they are paying for. Auction hunter first scrapes the data from online car auction postings, saves the information in a database, and display the best auctions to the user though a web interface. There is a component that helps to analyze the information available, and predict which auctions are the highest value. This sends information back into the database to be used by the Web interface.

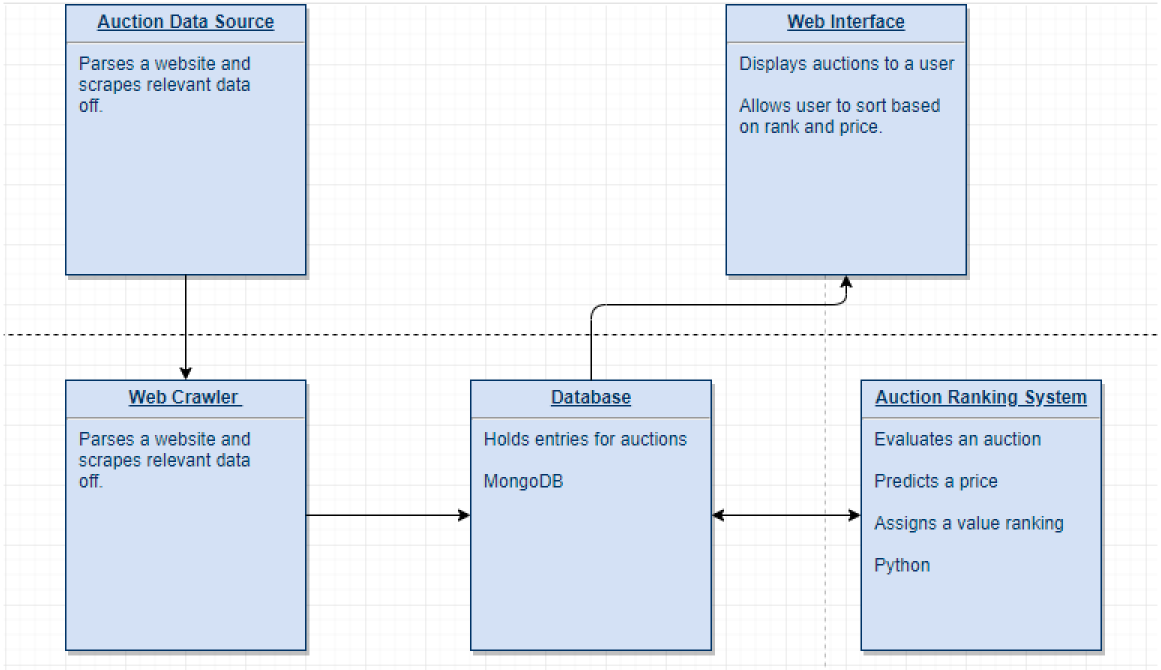
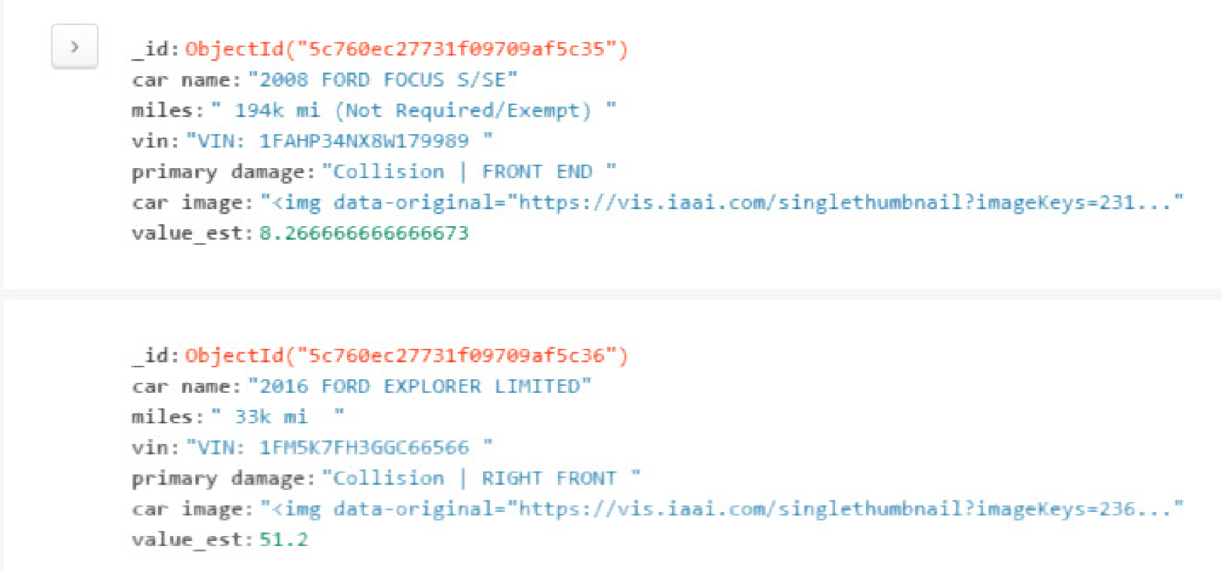

Auction Data Source
Parses a website and scrapes relevant data off.
Web Interface
Displays auctions to a user
Allows user to sort based on rank and price.
Web Crawler
Parses a website and scrapes relevant data off.
Database
Holds entries for auctions
MongoDB
Auction Ranking System
Evaluates an auction
Predicts a price
Assigns a value ranking
Python

Fig. 1. Flow Design

2 Progress

Our Auction Hunter program currently scrapes entries from IAAI, stores them in a database, performs value calculations, and displays to a user through a web interface.

To perform the web scraping, we are using the 'Scrapy' tool, which interfaces directly with our our database. We simply had to modify a pipeline to send the auction entries to mongoDB instead of saving to a file. To perform the value calculation, we use the mileage and damage information to skew the value of the car in one direction or another. This is a naive implementation that will need to be improved and tested. One of the biggest precursors to improving the value estimation is data availability. We are currently hosting our database on a personal server, which means we can all access the database from home. To make viewing the contents of the database easier, we are using mongoDB's GUI, called 'compass'. Below is a screenshot of our current database entries. [1]



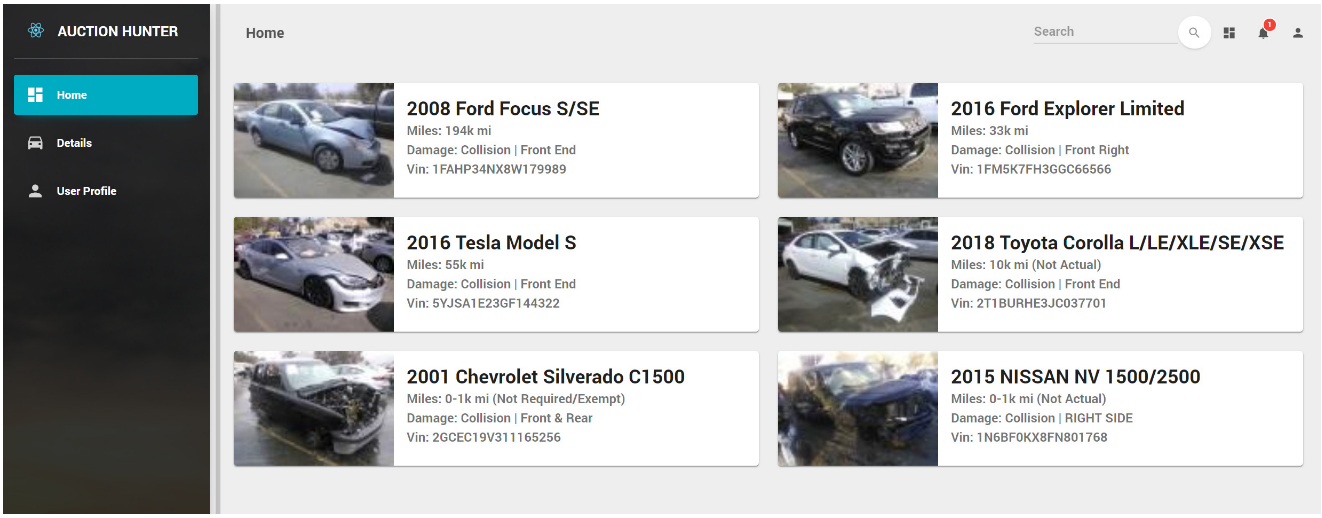
```
_id: ObjectId("5c760ec27731f09709af5c35")
car name: "2008 FORD FOCUS S/SE"
miles: " 194k mi (Not Required/Exempt) "
vin: "VIN: 1FAHP34NX8W179989 "
primary damage: "Collision | FRONT END "
car image: "<img data-original="https://vis.iaai.com/singlethumbnail?imageKeys=231..."
value_est: 8.266666666666673

_id: ObjectId("5c760ec27731f09709af5c36")
car name: "2016 FORD EXPLORER LIMITED"
miles: " 33k mi  "
vin: "VIN: 1FM5K7FH3GGC66566 "
primary damage: "Collision | RIGHT FRONT "
car image: "<img data-original="https://vis.iaai.com/singlethumbnail?imageKeys=236..."
value_est: 51.2
```

Fig. 2. Database entries

2.1 Web Interface Progress

Auction Hunter's web site is comprised of two main parts: the frontend and the backend. The backend is hosted on the server and serves the frontend to the visitor. The frontend is downloaded onto their computer and then run in their web browser. The frontend then makes REST API calls to the backend to interact with the database. REST is a widely used API architecture that defines and formalizes the interaction between two systems. The frontend uses the REST API to makes various types of calls including logging in a user, updating settings, requesting auction data, etc. The backend is written in Python using a library called Django. Using Django, Auction Hunter can handle logins, modify the database, and send emails and other notifications. The frontend is written in JavaScript using the React framework to generate the layout and user interaction with the server. Users of Auction Hunter will only ever interact directly with the frontend, which then will translate the necessary actions into REST commands to send to the server.



2008 Ford Focus S/SE
Miles: 194k mi
Damage: Collision | Front End
Vin: 1FAHP34NX8W179989

2016 Ford Explorer Limited
Miles: 33k mi
Damage: Collision | Front Right
Vin: 1FM5K7FH3GGC66566

2016 Tesla Model S
Miles: 55k mi
Damage: Collision | Front End
Vin: 5YJSA1E23GF144322

2018 Toyota Corolla L/LE/XLE/SE/XSE
Miles: 10k mi (Not Actual)
Damage: Collision | Front End
Vin: 2T1BURHE3JC037701

2001 Chevrolet Silverado C1500
Miles: 0-1k mi (Not Required/Exempt)
Damage: Collision | Front & Rear
Vin: 2GCEC19V311165256

2015 NISSAN NV 1500/2500
Miles: 0-1k mi (Not Actual)
Damage: Collision | RIGHT SIDE
Vin: 1N6BF0KX8FN801768

Fig. 3. HomePage of Auction Hunter

Shown above is the front page to our website. It shows the six most recent cars added to the database along with some basic details about the cars. You can then click on the car and brings you to a more detailed page about the auction its self. There is also a search bar that does a basic keyword search on the car.

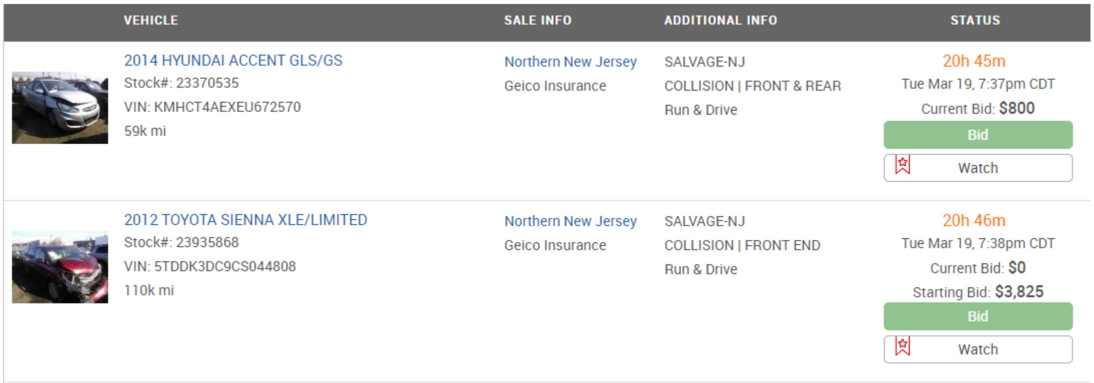
2.2 Web Scraper Progress

When customers are trying to buy a salvage car and they will want to see the prices at different platforms at a single place. In such situations, Web crawler will full play its function on building an aggregator platform.

IAAI.com is a website that sells salvage cars. [2] It will be our test site, and we are going to scrape Timed Auctions for salvage cars. The first step is to generate a basic spider using python:

```
scrapy crawl iaaibot
```

That is what the IAAI Timed Auctions page looks like:



VEHICLE	SALE INFO	ADDITIONAL INFO	STATUS
2014 HYUNDAI ACCENT GLS/GS Stock#: 23370535 VIN: KMHCT4AEXEU672570 59k mi	Northern New Jersey Geico Insurance	SALVAGE-NJ COLLISION | FRONT & REAR Run & Drive	20h 45m Tue Mar 19, 7:37pm CDT Current Bid: $800 Bid Watch
2012 TOYOTA SIENNA XLE/LIMITED Stock#: 23935868 VIN: 5TDDK3DC9CS044808 110k mi	Northern New Jersey Geico Insurance	SALVAGE-NJ COLLISION | FRONT END Run & Drive	20h 46m Tue Mar 19, 7:38pm CDT Current Bid: $0 Starting Bid: $3,825 Bid Watch

Fig. 4. Timed Auctions Page

From this page, the following data about a salvage car needs to be extracted:

- Salvage car name
- Salvage car miles
- Salvage car vin
- Salvage car primary damage
- Salvage car image's url

Extracting the URLs include salvage car photo:

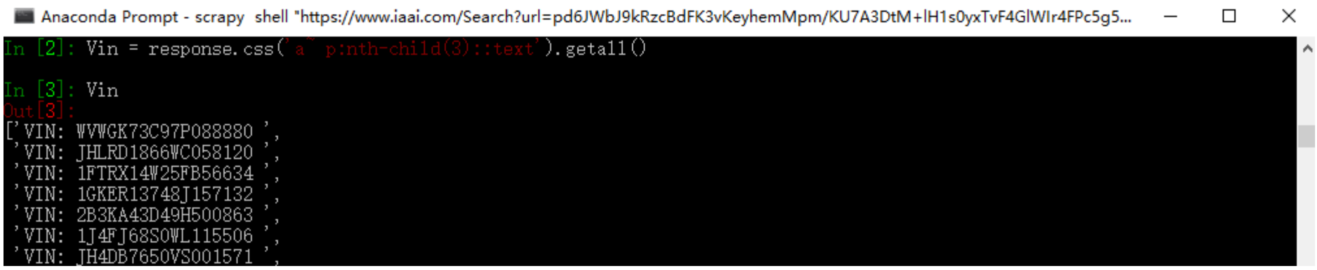
After searching through the web page source code by using the developer tools of Chrome and Google extension SelectorGadget. We found that the car photos are stored under vis.iaai.com with ”imageKeys” and their CSS labels are ”.lazy”.

The CSS labels ”.lazy” can be used to extract image URLs.

```
CarImage = response.css('.lazy').getall()
```

Extracting the URLs include salvage car Vin:

We can use a similar method to find VIN's location which is an attribute of the <a p:nth-child(3)>tag.



Anaconda Prompt - scrapy shell "https://www.iaai.com/Search?url=pd6JWbJ9kRzcBdFK3vKeyhemMpm/KU7A3DtM+lH1s0yxTvF4GlWIr4FPc5g5...

```
In [2]: Vin = response.css('a~ p:nth-child(3)::text').getall()

In [3]: Vin
Out[3]:
['VIN: WVWGK73C97P088880 ',
 'VIN: JHLRD1866WC058120 ',
 'VIN: 1FTRX14W25FB56634 ',
 'VIN: 1GKER13748J157132 ',
 'VIN: 2B3KA43D49H500863 ',
 'VIN: 1J4FJ68S0WL115506 ',
 'VIN: JH4DB7650VS001571 ',
```

Fig. 5. Scraping Vin

```
Vin = response.css('a~ p:nth-child(3)::text').getall()
```

Extracting the URLs include salvage car name, miles, and primary damage:

Similarly, we are able to extract the name, miles and primary damage or any other necessary data of salvage car.

```
#Extracting the content using css selectors
Vin = response.css('a~ p:nth-child(3)::text').getall()
```

Time to download the extracted photos of salvage car:

As mentioned in Design Document, Scrapy provides the images pipelines: once we got the data from website, we are able to pass them through different pipelines. By taking advantage of this function, the image pipeline allows us to download extracted photos of salvage car. In addition, we are able to convert the format of images and generate thumbnail.

```
#enable the images pipeline.
SalvageCarPhotos_PIPELINES='scrapy.pipelines.images.ImagesPipeline': 1
#set the local download address.
Photo_Store = 'Users/Desktop/SalvageCarPhotos/'
#Generate two kinds of thumbnail for each salvage car photo, one small, one large.
GenerateThumbnail = {'small': (20, 20), 'large': (100, 100),}
```

3 Future Plans

3.1 Web Interface Advancements

- Once database has more data scrapped from auction site, display more in detail page.
- Make logging in easier by displaying better errors when the username or password is incorrect.
- Finish allowing the user to set and recieve alerts about auctions.

3.2 Database Advancements

- Database will need to be expanded to hold more elements, and more attributes from each element.
- Value estimations will need to take into account the additional elements.
- Value estimations will have to be studied and tested more thoroughly. This will include creating trend visualizations, and compare to the current bid price to find where it overvalues/undervalues auctions.

3.3 Web Crawler Advancements

- Instead of just scraping the stuff from the current page, we want quotes from all the pages in the website. The corresponding solution might be that we could get the anchor element like ”next” or ”next page” to set a for loop to keep the crawling going through all the pages on IAAI.
- We only have extracted the url of car images rather than the exact images. We are looking for a method to download the extracted photos of salvage car to local. The corresponding solution includes using the images pipeline in Scrapy.
- We are working on consolidating the extracted data of salvage car into our database. The corresponding solution is that we need to extract data with a better logic. For example, removing all redundancies.

4 PROBLEMS

One issues that we are running into is the difficulty in updating our database in real time to match the original source. New auctions are created and removed every day, and our database will have to reflect that in order to stay relevant. Our current implementation doesn’t allow for routine updates, meaning our database can become out of sync quickly.

Another problem faced along the way was implementing features that other components cannot use effectively. This happened when a handful of database features were implemented for adding and reading entries. We later found out that our web scraper was able to interface with the database automatically, completely bypassing the work that was completed on the database. Although this is a much easier solution, it resulted in some wasted implementation time. This may have been something that could have been avoided if more research was done into how each component will work together before implementation started. One benefit was that initially going about this the hard way made it easier to implement future database advancements, where existing code could be salvaged.

5 CODE SNIPPETS

5.1 Web Scraper

The following snippet demonstrates how the web scraper communicates with the database, which is called the ”mongoDB pipeline”.

```
def open_spider(self, spider):
    self.client = pymongo.MongoClient(self.mongo_uri)
    self.db = self.client[self.mongo_db]

def process_item(self, item, spider):
    self.db[self.collection_name].insert_one(dict(item))
    return item
```

Fig. 6. Calculate value estimation

Scrapy will call open_spider before it begins scraping, which establishes its connection to the mongo database. Whenever it processes an item, it inserts that item into the database.

5.2 Database and Value Calculation

The below code snippet is how value calculations are added to all the available entries in the database. It may seem logical to instead calculate these values when the database is first populated. However, Scrapy makes it very easy to insert directly into the database, so it is easier to populate automatically, then sanitize and enhance the database later on.

```
#return a cursor object of all entries with a certain year.
result_cursor = db.collection.find( { year: { "gt:" int(year) } } )
#Find the total number returned
count = result_cursor.collection.count_documents({})
#Parse through all returned entries
for x in range(count):
    #Go to the next (this will point to the first entry after the first call)
    current_entry = result_cursor.next()
    #Get value estimation number
    value = enhancer.getValueEstimation(current_entry)
    currentVin = current_entry.get("vin")
    #Add the value estimation to the database as a new attribute
    database.scrapy_items.update_one({"vin": currentVin}, {"\$set": {"value_est": value}})
```

Fig. 7. Adding value calculation to database

The current value estimation is very simplistic. This is chosen more as a initial best guess, and will be improved as more data is available. Currently, the value estimation is only based on the damage and miles. The ’damage’ variable is calculated based on the severity of the damage description. This ranges from 0-5. These values are weighted, by ’milesWeight’ and ’damageWeight’ which both affect how much that calculation changes the value. These configuration variables are defined before any estimation is performed.

```
#Miles in thousands
value -= ((miles*2)/150.0 - 1)*self.milesWeight
#Damage from 0 to 5(most impactful damage)
value -= damage*(self.damageWeight/2.0)
```

Fig. 8. Calculate value estimation

5.3 Web Interface

The most important part of the web interface is the displaying of data from the database. The code below shows how a single row from the database represents a car and then that car’s data is then displayed nicely to the user, using HTML and CSS. The return function has to return all of the cars the home page needs to display. We created an inline function to create a single auction card which is a library we created to display a single auction to the user. The map function takes each car in a list and returns a list of HTML formatted code that is then returned to the main function to display.

```
return (
  <div><Grid container spacing={24}>
    {this.cars.map((car) => {
      return ( <Grid item xs={6}>
                 <AuctionCard
                   carImage={car.carImage}
                   carName={car.carName}
                   miles={car.miles}
                   vin={car.vin}
                   damage={car.damage}
                 />
               </Grid>
             )
         })} </Grid></div>); })
```

Fig. 9. web

References

[1] MongoDB, “What is mongodb,” https://www.mongodb.com/what-is-mongodb, 2018, accessed: 2018-11-9.

[2] IAAI, “Faq,” https://www.iaai.com/Support/SupportFaq.aspx, 2018, accessed: 2018-11-9.