Hearthstone Cluster Project - KMeans Clustering

This project aims to use clustering algorithms on the Hearthstone decks data from Hearthpwn. In this notebook, the clustering algorithm of choice is KMeans Clustering.
The data is obtained from history of hearthstone hosted on Kaggle [here (https://www.kaggle.com/romainvincent/history-of-hearthstone)](https://www.kaggle.com/romainvincent/history-of-hearthstone)
History of Hearthstone dataset is scraped by the Kaggle user "romainvincent" roughly two years ago.

Introduction and background

Hearthstone is an online trading card game where players build a deck of 30 cards and face each other. The quality of the deck heavily depends on the synergy of the cards included within the deck. Various classes and synergies availble at the players' disposal gives rise to numerous deck archetypes. These archetypes are essentially different types of decks with different win conditions or the gameplan to defeat the opponent. The cards that synergize with each other form a group of cards that if they were to be included in a deck, they have to be included togehter. This is generally known as a **pcakage**.
This notebook aims to explore and surface these synergies through the decklist dataset.

Approach and methodology

Each row of the data is a deck that has been posted on hearthpwn. For each deck, there are 30 cards and their card ID ("dbfId") are recorded under the columns "card_0" to "card_29". First, the 30 card columns have to be combined into one single column with all the 30 IDs of the cards within a deck. Then, a column is created for each card. If a deck contains a particular card, the value under the column of that particular card will the count of the copies of the card within the deck.
For example, if deck number 10 has 2 copies of card 111, the value under the column "111" for deck 10 will be 2.

Importing packages

In [1]:
```
from collections import Counter
import json
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sns
```

Loading data

In [2]:
```
refs = json.load(open("./history-of-hearthstone/refs.json"))
decks_data = pd.read_csv("./history-of-hearthstone/data.csv")
```

In [3]:
```
# taking a look at the head
decks_data.head()
```

Out[3]:

	craft_cost	date	deck_archetype	deck_class	deck_format	deck_id	deck_set	deck_type	ra
0	9740	2016-02-19	Unknown	Priest	W	433004	Explorers	Tavern Brawl	
1	9840	2016-02-19	Unknown	Warrior	W	433003	Explorers	Ranked Deck	
2	2600	2016-02-19	Unknown	Mage	W	433002	Explorers	Theorycraft	
3	15600	2016-02-19	Unknown	Warrior	W	433001	Explorers	None	
4	7700	2016-02-19	Unknown	Paladin	W	432997	Explorers	Ranked Deck	

5 rows × 41 columns

Data types of the columns

Column card_0 to card_29 are of the same type - int.

In [4]:
```
decks_data.dtypes[1:13]
```

Out[4]:
```
date              object
deck_archetype    object
deck_class        object
deck_format       object
deck_id            int64
deck_set          object
deck_type         object
rating             int64
title             object
user              object
card_0             int64
card_1             int64
dtype: object
```

The column "date" is not of the correct type. Changing the column datatype

In [5]:
```
decks_data['date'] = pd.to_datetime(decks_data['date'])
```

In [6]:
```python
decks_data.dtypes[1:13]
```

Out[6]:
```
date                 datetime64[ns]
deck_archetype               object
deck_class                   object
deck_format                  object
deck_id                       int64
deck_set                     object
deck_type                    object
rating                        int64
title                        object
user                         object
card_0                        int64
card_1                        int64
dtype: object
```

In [7]:
```python
decks_data.isnull().sum()[decks_data.isnull().sum() != 0]
```

Out[7]:
```
title    8
dtype: int64
```

There are 8 decks without title. Filling the null titles with a placeholder name

In [8]:
```python
decks_data = decks_data.fillna('some deck')
```

Exploring the number of decks by type and format

Total number of decks

In [9]:
```python
decks = len(decks_data)
print ('Number of decks :', decks)
```

```
Number of decks : 346232
```

Total wild decks

In [10]:
```python
wild = len(decks_data[decks_data['deck_format'] == 'W'])
print ('Wild decks:', wild)
```

```
Wild decks: 175446
```

Sum of decks by type

In [11]:
```
decks_data['deck_type'].value_counts()
```

Out[11]:
```
Ranked Deck      202375
None              91058
Theorycraft       19688
Arena             14095
PvE Adventure      9059
Tavern Brawl       6360
Tournament         3597
Name: deck_type, dtype: int64
```

Sum of *standard* decks by type

In [12]:
```
decks_data[decks_data['deck_format'] != 'W']['deck_type'].value_counts()
```

Out[12]:
```
Ranked Deck      101431
None              47837
Theorycraft        9644
Arena              7233
Tournament         2460
PvE Adventure      1398
Tavern Brawl        783
Name: deck_type, dtype: int64
```

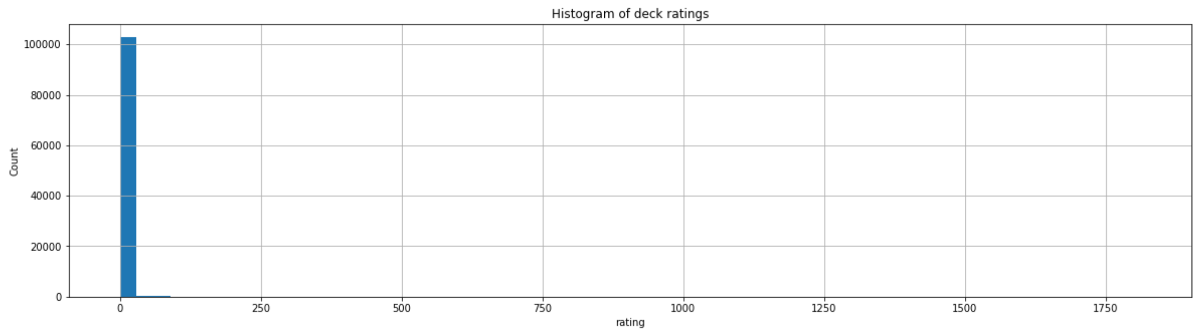
Choosing only *standard* ranked and tournament decks.

In [13]:
```
standard_decks = decks_data[decks_data['deck_format'] == 'S']
standard_decks = standard_decks[standard_decks['deck_type'].isin(['Ranked Dec
k', 'Tournament'])]
standard_decks['deck_type'].value_counts()
```

Out[13]:
```
Ranked Deck    101431
Tournament       2460
Name: deck_type, dtype: int64
```

Histogram of the rating

In [14]:
```
plt.figure(figsize=(20,5))
plt.hist(standard_decks['rating'], bins=60)
plt.title("Histogram of deck ratings")
plt.ylabel("Count")
plt.xlabel("rating")
plt.grid(True)
plt.show()
```



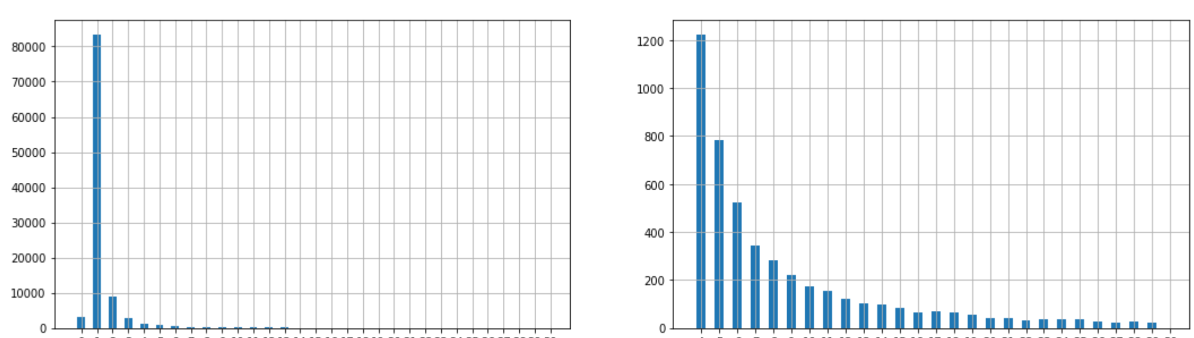
Decks with very low rating absolutely dominates the population. These decks are unlikely to be representative of the good decks that utilize package synergy too. As such, decks with low ratings are to be filtered.

Examining the decks with low rating

(rating between 0-30)

In [15]:
```
low_ratings = standard_decks[standard_decks['rating'] < 31]
print(low_ratings['deck_type'].value_counts())

f, (ax1, ax2) = plt.subplots(1, 2, sharey=False, figsize=(18, 5))
ax1.hist(low_ratings['rating'], bins=np.arange(31)-0.25, width=0.5)
ax1.grid()
ax1.set_xticks(range(0,31));

ax2.hist(low_ratings[low_ratings['rating'] > 3]['rating'], bins=np.arange(4,31
)-0.25, width=0.5)
ax2.grid()
ax2.set_xticks(range(4,31));
```

```
Ranked Deck    100472
Tournament       2431
Name: deck_type, dtype: int64
```



The population of decks appear to be less skewed at rating value > 20. Hence, only decks with rating > 20 are considered in this study.
Looking at rated decks' columns.

Creating the "card" columns

In [16]:
```
rated_decks = standard_decks[standard_decks['rating'] > 20].copy()
rated_decks.columns
```

Out[16]:
```
Index(['craft_cost', 'date', 'deck_archetype', 'deck_class', 'deck_format',
       'deck_id', 'deck_set', 'deck_type', 'rating', 'title', 'user', 'card_
0',
       'card_1', 'card_2', 'card_3', 'card_4', 'card_5', 'card_6', 'card_7',
       'card_8', 'card_9', 'card_10', 'card_11', 'card_12', 'card_13',
       'card_14', 'card_15', 'card_16', 'card_17', 'card_18', 'card_19',
       'card_20', 'card_21', 'card_22', 'card_23', 'card_24', 'card_25',
       'card_26', 'card_27', 'card_28', 'card_29'],
      dtype='object')
```

First, the card IDs from card_0 to card_29 need to be combined into one space separated sentence to later generate the card counts for each card in a deck.

In [17]:
```
# Adding the first card to the "card" column. The first card is card_0.
rated_decks.loc[:, 'card'] = rated_decks.loc[:, 'card_0'].astype(int).astype(s
tr)

# for card_1 to card_29, concatenate the card ID to the existing 'card' colum
n.
for i in range(1,30):
    colname = 'card_' + str(i)
    rated_decks.loc[:, 'card'] = rated_decks.loc[:, 'card'] + ' ' + rated_deck
s[colname].astype(int).astype(str)
```

In [18]:

```
# checking the head of the dataframe
rated_decks.head()
```

Out[18]:

	craft_cost	date	deck_archetype	deck_class	deck_format	deck_id	deck_set	deck_type
137	8120	2016-02-19	Miracle Rogue	Rogue	S	432773	Explorers	Ranked Deck
1690	8180	2016-04-21	Ramp Druid	Druid	S	478188	Explorers	Ranked Deck
2414	1680	2016-04-30	Unknown	Warrior	S	520990	Old Gods	Ranked Deck
3527	1860	2016-04-30	Midrange Hunter	Hunter	S	520585	Old Gods	Ranked Deck
3992	8260	2016-06-27	Yogg Druid	Druid	S	579170	Old Gods	Ranked Deck

5 rows × 42 columns

In [19]:
```
# column "card" generated successfully. Dropping card_0 to card_29 columns.
rated_decks.drop(['card_0',
       'card_1', 'card_2', 'card_3', 'card_4', 'card_5', 'card_6', 'card_7',
       'card_8', 'card_9', 'card_10', 'card_11', 'card_12', 'card_13',
       'card_14', 'card_15', 'card_16', 'card_17', 'card_18', 'card_19',
       'card_20', 'card_21', 'card_22', 'card_23', 'card_24', 'card_25',
       'card_26', 'card_27', 'card_28', 'card_29'], axis=1, inplace=True)
rated_decks.head()
```

Out[19]:

	craft_cost	date	deck_archetype	deck_class	deck_format	deck_id	deck_set	deck_type
137	8120	2016-02-19	Miracle Rogue	Rogue	S	432773	Explorers	Ranked Deck
1690	8180	2016-04-21	Ramp Druid	Druid	S	478188	Explorers	Ranked Deck
2414	1680	2016-04-30	Unknown	Warrior	S	520990	Old Gods	Ranked Deck
3527	1860	2016-04-30	Midrange Hunter	Hunter	S	520585	Old Gods	Ranked Deck
3992	8260	2016-06-27	Yogg Druid	Druid	S	579170	Old Gods	Ranked Deck

The column "card" now contains 30 card IDs for the cards within the deck.

Cards details from JSON

In [20]:
```
card_data = pd.DataFrame.from_records(refs)

# dropping cards without dbfId
card_data = card_data[~card_data['dbfId'].isna()]

# casting the dbfId to integer instead of float
card_data['dbfId'] = card_data['dbfId'].astype(int)

card_data.head()
```

Out[20]:

	artist	attack	cardClass	classes	collectible	collectionText	cost	dbfId	durability	elite
0	Jakub Kasper	4.0	NEUTRAL	NaN	True	NaN	4.0	2518	NaN	NaN
1	NaN	4.0	NEUTRAL	NaN	NaN	NaN	6.0	1769	NaN	True
2	NaN	NaN	NEUTRAL	NaN	NaN	NaN	NaN	10081	NaN	NaN
3	Mauricio Herrera	3.0	WARRIOR	NaN	True	NaN	3.0	40569	NaN	NaN
4	Ittoku	2.0	NEUTRAL	NaN	True	NaN	4.0	1370	NaN	NaN

5 rows × 32 columns

In [21]:
```
# reading the column names
card_data.columns
```

Out[21]:
```
Index(['artist', 'attack', 'cardClass', 'classes', 'collectible',
       'collectionText', 'cost', 'dbfId', 'durability', 'elite', 'entourage',
       'faction', 'flavor', 'health', 'hideStats', 'howToEarn',
       'howToEarnGolden', 'id', 'mechanics', 'multiClassGroup', 'name',
       'overload', 'playRequirements', 'playerClass', 'race', 'rarity',
       'referencedTags', 'set', 'spellDamage', 'targetingArrowText', 'text',
       'type'],
      dtype='object')
```

In [22]:
```
# reading the shape, 3116 cards in total
card_data.shape
```

Out[22]:
```
(3116, 32)
```

In [23]:
```
# obtaining a list of all card IDs in the reference card data
card_ids = card_data['dbfId'].astype(str).tolist()
```

In [24]:
```
def contains(cardlist, card):
    '''
    A functiont to check how many times a card ID ("card")
    appear in the cardlist.
    '''
    cardcount = Counter(cardlist.split())[card]
    return cardcount
```

In [25]:
```
# testing the function

somestring = '77 77 90 90 175 175 315 315 395 395 447 447 555 555 564 564 587
 587 635 635 662 662 672 672 906 906 1004 1004 1084 1084'
Counter(somestring.split())['555']
```

Out[25]: 2

In [26]:
```
# for every card in the cardlist, check if it is present in the card column.
# Then add the count (0,1 or 2) onder the respective card's column

for i in (card_ids):
    rated_decks.loc[:,i] = rated_decks.loc[:,'card'].apply(lambda x: contains(
x,i))
```

In [27]:
```
# remove any card column that does not have a card present in the decks at al
l.
# That is, the value of the column is 0 for every row.

rated_decks = rated_decks.loc[:, (rated_decks != 0).any(axis=0)]
rated_decks.head()
```

Out[27]:

	craft_cost	date	deck_archetype	deck_class	deck_format	deck_id	deck_set	deck_type
137	8120	2016-02-19	Miracle Rogue	Rogue	S	432773	Explorers	Ranked Deck
1690	8180	2016-04-21	Ramp Druid	Druid	S	478188	Explorers	Ranked Deck
2414	1680	2016-04-30	Unknown	Warrior	S	520990	Old Gods	Ranked Deck
3527	1860	2016-04-30	Midrange Hunter	Hunter	S	520585	Old Gods	Ranked Deck
3992	8260	2016-06-27	Yogg Druid	Druid	S	579170	Old Gods	Ranked Deck

5 rows × 794 columns

In [28]:
```
# saving processed data
rated_decks.to_csv('processed_HS_data.csv')
```

Checking if all the cards in a deck add up.

In [29]:
```
rated_decks.loc[:, 'cardsaddup'] = rated_decks.iloc[:,12:].sum(axis=1) == 30
rated_decks['cardsaddup'].sum() == len(rated_decks)
```

Out[29]: True

Applying KMeans clustering on both decks and cards

The most straightforward method of clusterning appears to be KMeans clustering. Here, for clustering of decks, the number of clusters used is 20 which is an approximate. For clustering of cards, the n value used is 30. Fine tuning of the n value will be done in a later section.

In [30]:
```
from sklearn.cluster import KMeans
```

clustering the decks together

Here, KMeans clustering is used to cluster the decks together based on their card choices. The expected results would be group of similar decks and the corresponding cards they have in common.

In [31]:
```
# the array clusters_decks now contains the number of the cluster the deck bel
ongs to
clusters_decks = KMeans(n_clusters=20, random_state=7).fit_predict(rated_decks
.iloc[:,12:])
print(clusters_decks)
```

```
[ 1  4 14 ... 13  1  3]
```

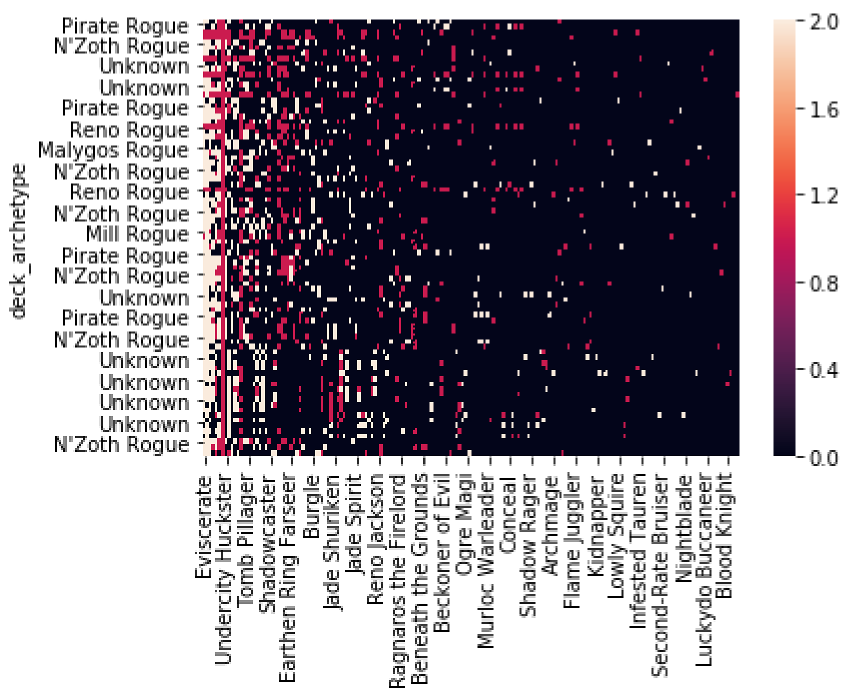
Visualizing the data

Heatmap appears to be the most intuitive way to visualize the data. The decks are on the y-axis. To improve the interpretability, the deck IDs are replaced by the deck archetypes. Deck archetypes are used instead of the deck names since they are more intuitive when it comes to describing type of a deck.
x-axis are the card names. The heatmap will be sorted in order of cards frequency. That is, the card that appears the most among the decks within the cluster will be on the leftmost of the axis.

In [32]:
```
# this lookup table will be used to translate card IDs to names
# the keys are integer. They have to be changed to string
card_lookup = card_data.set_index('dbfId')['name']
card_lookup = {str(k):v for k,v in card_lookup.items()}
```

In [33]:
```
# plotting the heat map of cluster number 1
cluster_number = 3
cluster_1 = rated_decks[clusters_decks == cluster_number].iloc[:,12:].astype(f
loat)

# rename the rows (index)
cluster_1.set_index([rated_decks[clusters_decks == cluster_number].iloc[:,2]],
inplace=True)

# rename the columns
cluster_1.rename(columns=dict(card_lookup), inplace=True)

# drop all clumns with no card present in the decks in the cluster
cluster_1 = cluster_1.loc[:, (cluster_1 != 0).any(axis=0)]

# sort the columns by their sum
cluster_1 = cluster_1.reindex(cluster_1.sum().sort_values(ascending=False).ind
ex, axis=1)

sns.heatmap(cluster_1)
```

Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x7ff176ad4a58>



Discussion

Here, we can see the all the Rogue decks are clustered together. Moreover, the main variant appears to be a N'Zoth Rogue. Two copies of strong Rogue cards, preferably with deathrattle, such as Eviscerate, Undercity Huckster, and Tomb Pillager are included in almost all the decks in this cluster.

clustering the *cards* together

Similar to clustering the decks done above, this section explores clustering of the cards using the decks as features. Through this, we can explore what cards frequently appear together and hence the cards that apear together can be considered a **package**.

In [34]:
```
# transpose the table

rated_decks_tpsd = rated_decks.drop(['cardsaddup'], axis=1)
rated_decks_tpsd = rated_decks_tpsd.iloc[:,12:]
rated_decks_tpsd = rated_decks_tpsd.T
print(rated_decks_tpsd.shape)

# creading a deck archetype lookup dictionary
deck_lookup = rated_decks['deck_archetype']
```

```
(782, 1296)
```

In [35]:
```
# changing the dbfId type form integer to string
card_data['dbfId'] = card_data['dbfId'].astype(int).astype(str)

# joining the card name on dbfId
rated_decks_tpsd = pd.merge(rated_decks_tpsd, card_data[['dbfId','name']], how
='left', left_index=True, right_on='dbfId')

# set the index to be the card name
rated_decks_tpsd.set_index('name', inplace=True)

# drop dfbId column
rated_decks_tpsd.drop(['dbfId'], axis=1, inplace=True)

rated_decks_tpsd.head()
```

Out[35]:

	137	1690	2414	3527	3992	4592	4965	5269	5327	5358	...	325935	325937	3
name														
Oasis Snapjaw	0	0	0	0	0	0	0	0	0	0	...	0	0	
Shadow Word: Death	0	0	0	0	0	0	0	0	0	0	...	0	0	
Silent Knight	0	0	0	0	0	0	0	0	0	0	...	0	0	
Unlicensed Apothecary	0	0	0	0	0	0	0	0	0	0	...	0	0	
Shadow Madness	0	0	0	0	0	0	0	0	0	0	...	0	0	

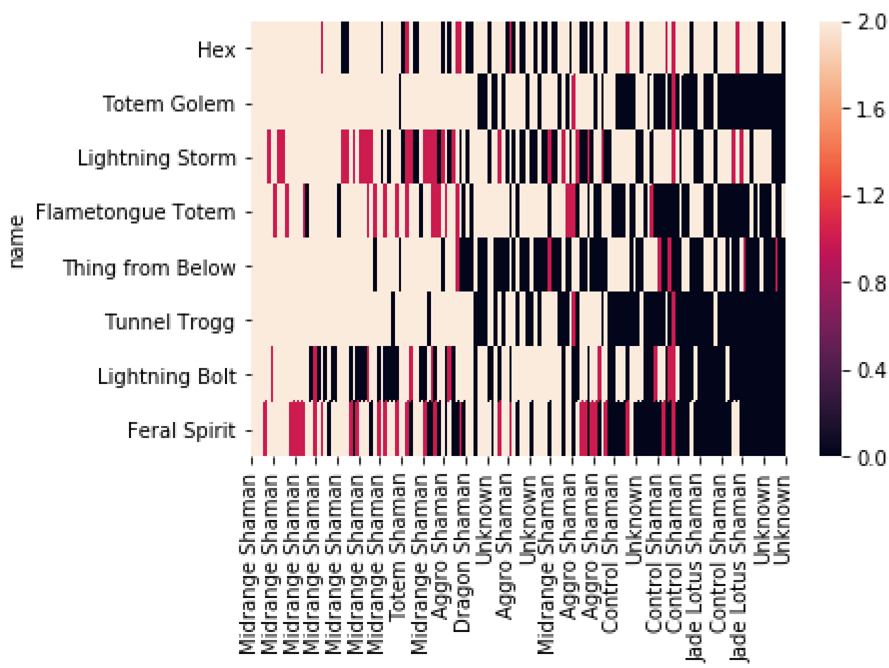
5 rows × 1296 columns

kmeans clustering

In [36]:
```
kmeansclf = KMeans(n_clusters=30, random_state=7)
clusters_cards = kmeansclf.fit_predict(rated_decks_tpsd)
```

In [37]:
```
def hs_heatmap(data,clusters,clusternumber):
    '''
    A function that produces a heatmap given the dataset, the KMeans model, an
d the predicted clusters.
    '''
    # subsetting the desired cluster
    card_cluster = data[clusters == clusternumber]#.astype(float)

    # sort the columns by their sum
    card_cluster = card_cluster.reindex(card_cluster.sum().sort_values(ascendi
ng=False).index, axis=1)
    card_cluster.rename(columns=dict(deck_lookup), inplace=True)

    # drop all clumns with no deck present in the decks in the cluster
    card_cluster = card_cluster.loc[:, (card_cluster != 0).any(axis=0)]

    # sort all rows by their sum
    sorted_index = card_cluster.sum(axis=1).sort_values(ascending=False)
    card_cluster = card_cluster.loc[sorted_index.index]

    sns.heatmap(card_cluster)
```

In [38]:
```
hs_heatmap(rated_decks_tpsd, clusters_cards, 3)
```



Discussion

The heatmap above shows that KMeans clustering is effective at grouping the cards that frequently appear in the same deck togehter. In the example above, the heatmap shows a **package** of Shaman cards. These are the cards that appear together mostly in Midrange Shaman decks. They are Hex, Totem Golem, Lightning Storm, Flametongue Totem, Thing from Below, Tunnel Trogg, Lightining Bolt and Feral Spirit. These cards synergize well together using the Totem tribe and overload effects.

Fine-tuning the model

To find the optimal n value, root mean squared distance is used to measure the tightness of a cluster. This root mean squared value is then used to measure how well the cards are clustered together.

In [39]:
```
def msqdist(data,classifier,classifier_output):
    '''
    Calculating the mean square distance for each cluster
    '''
    clusters = []
    intra_msqdist = []
    num_cards = []
    nclusters = classifier.get_params()['n_clusters']
    for n in range(0, nclusters):
        clsutercentroid = classifier.cluster_centers_[n]
        cards_in_cluster_n = data[classifier_output == n].copy()
        card_num = len(cards_in_cluster_n)
        # use the norm function to obtain the root squared distance
        cards_in_cluster_n.loc[:, 'dist-from-centroid'] = cards_in_cluster_n.a
pply(
            lambda x: np.linalg.norm(x - clsutercentroid), axis=1)
        # finding the mean value
        average_dist = np.mean(cards_in_cluster_n['dist-from-centroid'])
        clusters.append(n)
        intra_msqdist.append(average_dist)
        num_cards.append(card_num)
    return clusters, intra_msqdist, num_cards
```

In [40]:
```
clusters, avgdist, num_cards = msqdist(rated_decks_tpsd, kmeansclf, clusters_c
ards)
```

In [41]:
```
summary = pd.DataFrame({'clusters' : clusters, 'avgdist' : avgdist, 'num_card
s' : num_cards}).set_index('clusters')
summary.sort_values(by='avgdist').head(8)
```

Out[41]:

clusters	avgdist	num_cards
10	0.000000	1
22	0.000000	1
2	0.000000	1
20	0.000000	1
8	0.000000	1
21	3.968627	2
1	4.899031	591
0	5.678849	4

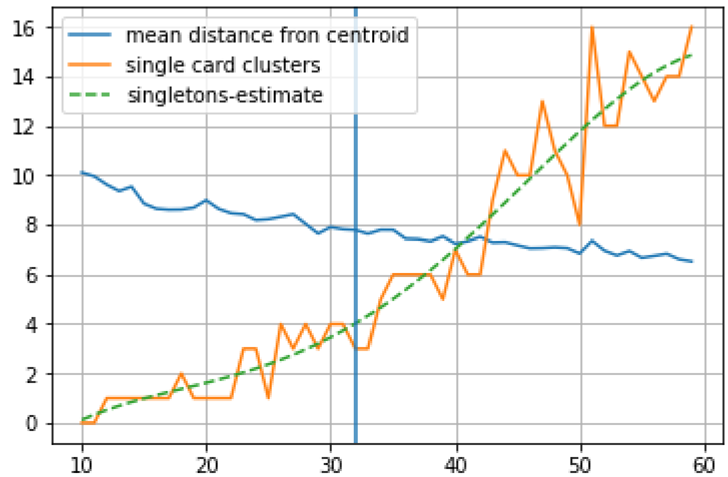
Beyond root mean suqared

As the number of clusters increase, the root mean squared distance decreases. Given the context of this project, **a package** refers to at least 2 cards working together. As such, even if the mean squared distance is low, a singleton cluster is not desirable. Therefore, the optimal model for this situation would be one where the root mean squared distance is low yet there is minimal number of singletons clusters.
Hence, we look for an "elbow" in the plot of the number of singletons against the number of KMeans cluster. That n value is then used as the optimal number of clusters.

In [42]:
```
k_vals = []
mean_dist = []
singletons = []
for k in range(10,60):
    kmeansclf = KMeans(n_clusters=k, n_init=30, random_state=7)
    clusters_cards = kmeansclf.fit_predict(rated_decks_tpsd)
    clusters, avgdist, num_cards = msqdist(rated_decks_tpsd, kmeansclf, cluste
rs_cards)
    summary = pd.DataFrame({'clusters' : clusters, 'avgdist' : avgdist, 'num_c
ards' : num_cards}).set_index('clusters')
    mean_dist_for_k = np.mean(summary[summary['num_cards'] >= 2]['avgdist'])
    singleton = len(summary[summary['num_cards'] == 1])
    k_vals.append(k)
    mean_dist.append(mean_dist_for_k)
    singletons.append(singleton)
    k_vals, mean_dist, singletons
```

In [43]:
```python
# fitting a smooth curve for the number of singletons.

singletons_fit = []
p = np.poly1d(np.polyfit(k_vals,singletons,5))
for i in k_vals:
    singletons_fit.append(p(i))
```

In [44]:
```python
plt.plot(k_vals, mean_dist, label='mean distance fron centroid')
plt.plot(k_vals, singletons, label='single card clusters')
plt.plot(k_vals, singletons_fit, linestyle='--', label='singletons-estimate')
plt.legend(loc='best')
plt.axvline(x=32)
plt.grid()
plt.show()
```



The optimal model

From above, the best value of n seems to be approximately 32.
Running the KMeans clustering one more time with n=32. Also, since **packages** are a handful of cards that are used together. The final result of this clustering exercise will be a list of **packages** with cards between 2 to 15 cards (half a deck).

In [45]:
```python
kmeansclf = KMeans(n_clusters=32, n_init=30, random_state=7)
clusters_cards = kmeansclf.fit_predict(rated_decks_tpsd)
```

In [46]:
```python
# obtaining the clusters with cards between 2 and 15
unique, counts = np.unique(clusters_cards, return_counts=True)
is_package = pd.Series(dict(zip(unique, counts)))
is_package = is_package[is_package <= 15]
is_package = is_package[is_package > 1]
is_package = np.array(is_package.index)
print(is_package)
```

```
[ 0  2  3  4  5  6  7  8 10 11 12 13 15 16 17 18 19 21 23 24 25 26 28 29
 30 31]
```

The above clusters are the clusters which can be defined to be a **package**.
Then, the card names are sorted according to the frequency of appearance within a cluster and tabulated below.

In [47]:

```
package_dict = {}
for i in is_package:
    cards_in_i = rated_decks_tpsd[clusters_cards == i]
    sorted_index = cards_in_i.sum(axis=1).sort_values(ascending=False)
    card_names = np.array(sorted_index.index)
    package_dict[i] = list(card_names)
package_df = pd.DataFrame.from_dict(package_dict,orient = 'index')
package_df.columns = ['card_1', 'card_2', 'card_3', 'card_4', 'card_5',
                      'card_6', 'card_7', 'card_8', 'card_9', 'card_10',
                      'card_11', 'card_12', 'card_13', 'card_14']
package_df.index.name = 'package'
package_df
```

Out[47]:

	card_1	card_2	card_3	card_4	card_5	card_6	
package							
0	Jade Spirit	Aya Blackpaw	Jade Idol	Jade Blossom	Jade Behemoth	None	
2	Innervate	Swipe	Wrath	Living Roots	None	None	
3	Animal Companion	Kill Command	Eaglehorn Bow	Quick Shot	Savannah Highmane	Unleash the Hounds	Explo
4	Fireball	Frostbolt	Arcane Intellect	Mana Wyrm	Arcane Missiles	Sorcerer's Apprentice	Arc
5	Power Word: Shield	Shadow Word: Death	Shadow Word: Pain	Northshire Cleric	Holy Nova	None	
6	Hex	Lightning Storm	Thing from Below	Mana Tide Totem	None	None	
7	Eviscerate	Backstab	SI:7 Agent	Fan of Knives	Sap	Gadgetzan Auctioneer	Pr
8	Totem Golem	Flametongue Totem	Tunnel Trogg	Lightning Bolt	Feral Spirit	Rockbiter Weapon	Flame
10	Sylvanas Windrunner	Entomb	Excavated Evil	Circle of Healing	Auchenai Soulpriest	Thoughtsteal	Caba
11	Acolyte of Pain	Fiery War Axe	Execute	Ravaging Ghoul	Slam	None	
12	Lava Shock	Elemental Destruction	Healing Wave	Ancestral Knowledge	Stormcrack	None	
13	Kor'kron Elite	Frothing Berserker	Blood To Ichor	None	None	None	
15	Bluegill Warrior	Tirion Fordring	Murloc Warleader	Forbidden Healing	Ivory Knight	Ragnaros, Lightlord	So
16	Bloodmage Thalnos	Flamestrike	Water Elemental	Forgotten Torch	Ice Block	Polymorph	Al
17	Shield Block	Shield Slam	Brawl	Bloodhoof Brave	Justicar Trueheart	Bash	
18	Wild Growth	Raven Idol	Nourish	Feral Rage	Mire Keeper	Fandral Staghelm	
19	Wild Pyromancer	Truesilver Champion	Consecration	Aldor Peacekeeper	Equality	None	
21	Maelstrom Portal	Spirit Claws	None	None	None	None	
23	Houndmaster	Fiery Bat	Call of the Wild	Infested Wolf	Kindly Grandmother	Huge Toad	Kir
24	Soulfire	Doomguard	Silverware Golem	Malchezaar's Imp	Darkshire Librarian	None	
25	Druid of the Claw	Power of the Wild	Violet Teacher	Savage Roar	Mark of Y'Shaarj	Druid of the Saber	Stra
26	Twilight Guardian	Blackwing Corruptor	Faerie Dragon	Blackwing Technician	Netherspite Historian	Book Wyrm	Twilig
28	Southsea Deckhand	N'Zoth's First Mate	Arcanite Reaper	Bloodsail Raider	Upgrade!	Dread Corsair	Hei

package	card_1	card_2	card_3	card_4	card_5	card_6	
29	Defender of Argus	Abusive Sergeant	Dark Peddler	Knife Juggler	Imp Gang Boss	Power Overwhelming	V
30	C'Thun's Chosen	Disciple of C'Thun	Beckoner of Evil	C'Thun	Twilight Elder	Twin Emperor Vek'lor	
31	Argent Squire	Argent Horserider	Keeper of Uldaman	Bilefin Tidehunter	Blessing of Kings	Divine Favor	Self

Conclusion

Looking at the table above, the packages identified by the KMeans clustering can be quite accurate. We can see popular packages such as the Freeze Mage (cluster number 16) or Zoo Warlock (cluster number 29). However, KMeans clustering does not allow for the same card to appear in different clusters. This, happends often in Hearthstone. As such, more sophisticated tools will have to be explored for further improve the accuracy of this **package** identificaiton exercise.

```
In [ ]:
```