In [1]:
```
# import modules
import pandas as pd
import numpy as np
import re
from bertopic import BERTopic
import random
import pickle
from sklearn.model_selection import train_test_split
from sklearn.metrics import multilabel_confusion_matrix
from sklearn.preprocessing import MultiLabelBinarizer
from sklearn.multiclass import OneVsRestClassifier
from sklearn.svm import SVC
from sklearn.neighbors import KNeighborsRegressor as KNN_reg
from sklearn.tree import DecisionTreeRegressor as DT_reg
from sklearn.ensemble import RandomForestRegressor as RF_reg
from sklearn import metrics
import matplotlib.pyplot as plt
from plotnine import *
import json
```

```
C:\Users\marya\miniforge3\envs\test_env\lib\site-packages\tqdm\auto.py:22: TqdmWarning: IProgress not found. Please update jupyt
er and ipywidgets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

In [2]:
```
# Load in data
(pd.read_feather('C:/Georgetown University/Courses/Spring Semester 2022/Text As Data/text-data-spr22/data/mtg.feather')# <-- wil
 .head(2)
)
```

Out[2]:

	color_identity	colors	converted_mana_cost	edhrec_rank	keywords	mana_cost	name	number	power	rarity	subtypes	supertypes	t
0	[W]	[W]	7.0	16916.0	[First strike]	[5, W, W]	Ancestor's Chosen	1	4.0	uncommon	[Human, Cleric]	[]	F str (T creat de com dama
1	[W]	[W]	5.0	14430.0	[Flying]	[4, W]	Angel of Mercy	2	3.0	uncommon	[Angel]	[]	Fly Wh Ange Me ent battle

In [3]:

```
# store full data
df = (pd.read_feather('C:/Georgetown University/Courses/Spring Semester 2022/Text As Data/text-data-spr22/data/mtg.feather')
)

# check shape
df.shape
```

Out[3]:

```
(56366, 20)
```

Part 1: Unsupervised Exploration

Investigate the BERTopic documentation (linked), and train a model using their library to create a topic model of the flavor_text data in the dataset above.

- In a topic_model.py, load the data and train a bertopic model. You will save the model in that script as a new trained model object
- add a "topic-model" stage to your dvc.yaml that has mtg.feather and topic_model.py as dependencies, and your trained model as an output
- load the trained bertopic model into your notebook and display
  - the topic_visualization interactive plot see docs

- Use the plot to come up with working "names" for each major topic, adjusting the number of topics as necessary to make things more useful.
- Once you have names, create a Dynamic Topic Model by following their documentation. Use the release_date column as timestamps.
- Describe what you see, and any possible issues with the topic models BERTopic has created. This is the hardest part... interpreting!

In [4]:
```
# load trained BERTopic model
topic_model = BERTopic.load("flav_text_model")
```

In [5]:
```
# access frequent topics
topic_model.get_topic_info()
```

Out[5]:

	Topic	Count	Name
0	-1	6519	-1_your_but_you_our
1	0	211	0_phyrexia_phyrexians_phyrexian_phyrexias
2	1	205	1_sword_steel_blade_swords
3	2	203	2_kami_kamigawa_observations_akki
4	3	182	3_goblins_goblin_demise_rivaled
...	...	...	...
944	943	10	943_demon_griselbrand_ereboss_skirsdag
945	944	10	944_dragonlings_spiraled_conjuring_meditate
946	945	10	945_overtaken_olanti_muraganda_sympathize
947	946	10	946_ambition_atrocities_bontu_paved
948	947	10	947_feature_strongest_playing_eyes

949 rows × 3 columns

-1 refers to all outliers and should typically be ignored. Next, let's take a look at the most frequent topic that was generated, topic 0:

In [6]:
```
topic_model.get_topic(0)
```

Out[6]:
```
[('phyrexia', 0.043827843001602675),
 ('phyrexians', 0.02688336867428565),
 ('phyrexian', 0.02091610171468328),
 ('phyrexias', 0.01766228423409125),
 ('vorinclex', 0.017132780867312482),
 ('mycosynth', 0.016803947198307186),
 ('azaxazog', 0.011714696504875105),
 ('thane', 0.011380103133789654),
 ('onetime', 0.010583000250677366),
 ('occurrence', 0.010250359441765718)]
```

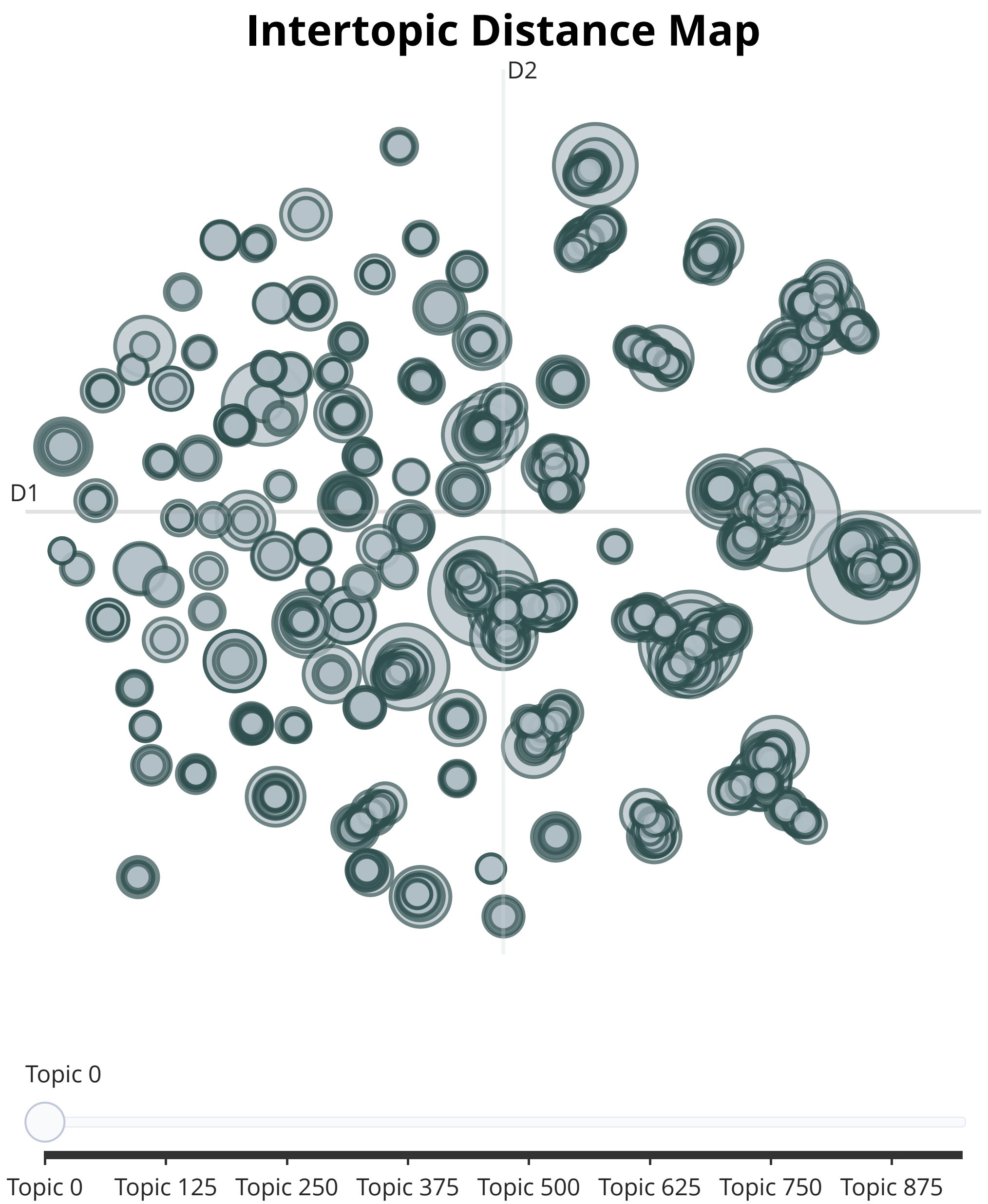
In [7]:
```
# store topic frequency
freq_topics = topic_model.get_topic_info().iloc[1: , :] # remove row with outliers (where Topic = -1)

# view percentiles of Count/frequency
freq_topics.Count.quantile([0.25,0.5,0.75,0.99])
```

Out[7]:
```
0.25    14.00
0.50    19.00
0.75    28.00
0.99    96.59
Name: Count, dtype: float64
```

Will select topics whose Count is in the 99th percentile.

Interactive plots

In [8]:
```
# visualize all topics
topic_model.visualize_topics()
```



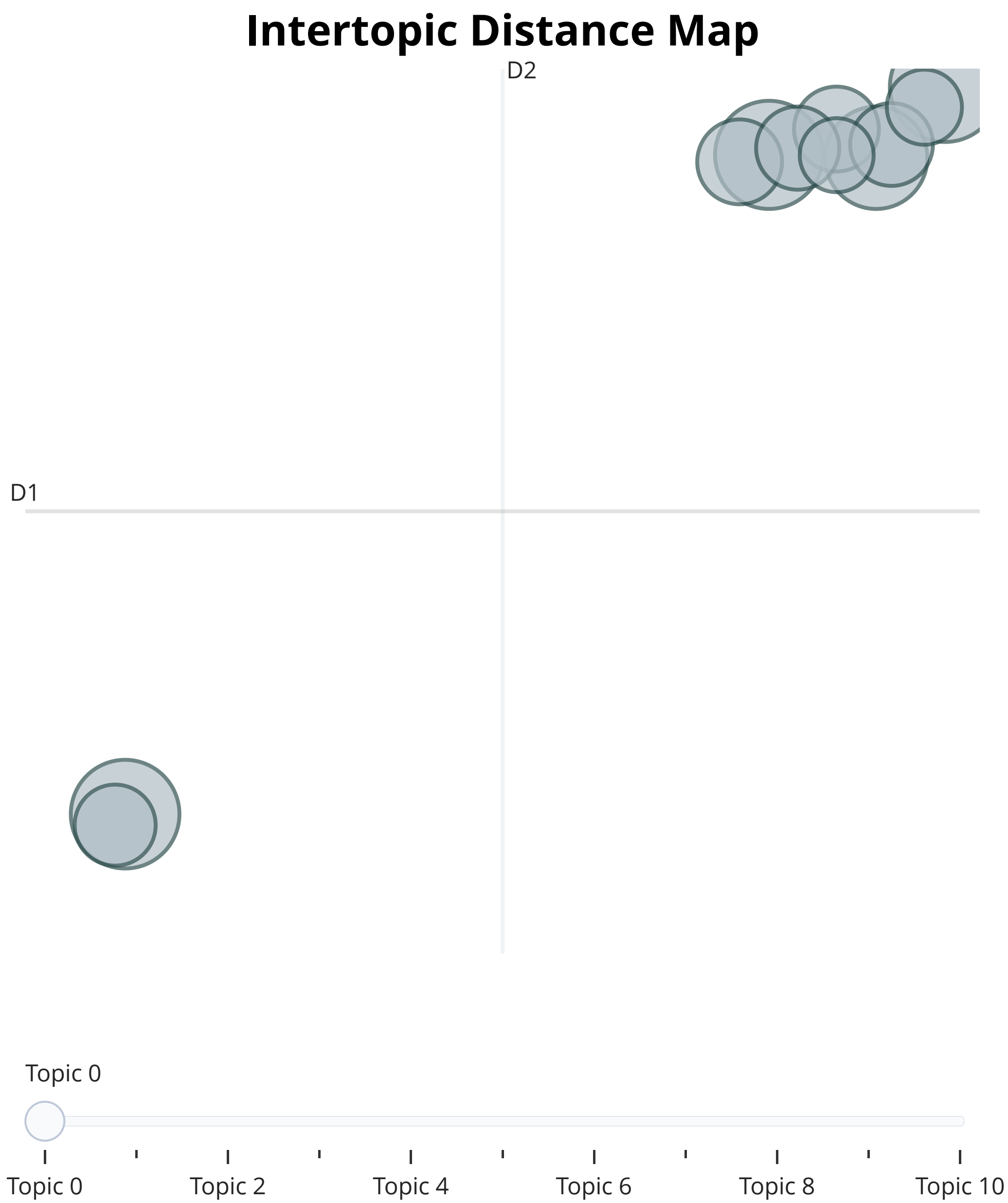
It's very hard to interpret 800+ topics, so I am going to select and visualize topics that have a frequency in the top percentile. Assumption: high frequency topics are representative of the main 'topic clusters'.

In [9]:
```
# view topics with freq in the top percentile
freq_topics.loc[freq_topics.Count > freq_topics.Count.quantile(0.99)]
```

Out[9]:

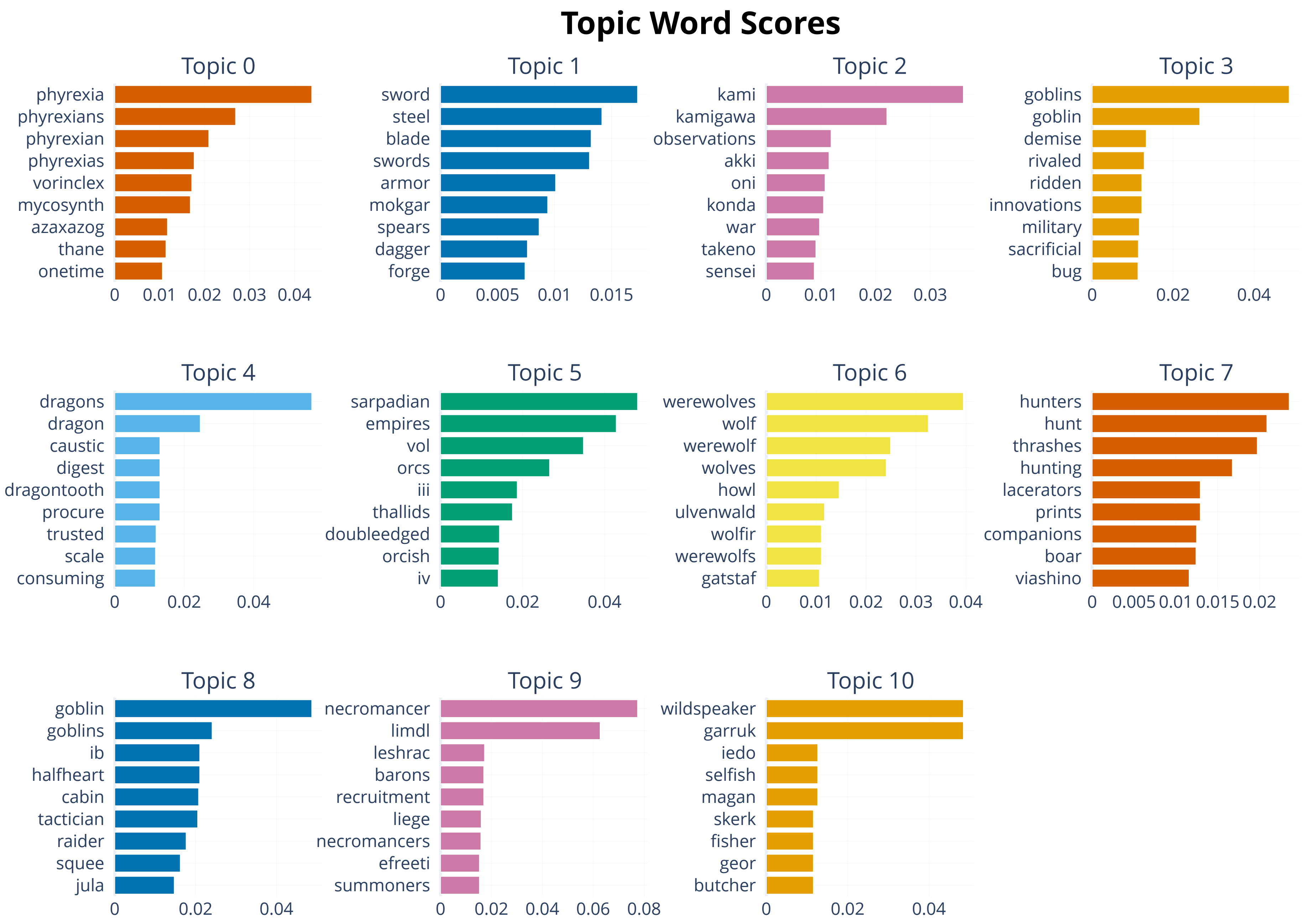
	Topic	Count	Name
1	0	211	0_phyrexia_phyrexians_phyrexian_phyrexias
2	1	205	1_sword_steel_blade_swords
3	2	203	2_kami_kamigawa_observations_akki
4	3	182	3_goblins_goblin_demise_rivaled
5	4	125	4_dragons_dragon_caustic_digest
6	5	125	5_sarpadian_empires_vol_orcs
7	6	120	6_werewolves_wolf_werewolf_wolves
8	7	119	7_hunters_hunt_thrashes_hunting
9	8	114	8_goblin_goblins_ib_halfheart
10	9	98	9_necromancer_limdl_leshrac_barons

In [10]:
```
# view intertopic distance map
topic_model.visualize_topics(topics = [-1,0,1,2,3,4,5,6,7,8,9,10])
```



In order to name these topics, I will visualize them as bar charts that include the top 9 words in each topic. (I tried including the top 10 words but doing that only displays alternate written words which makes it difficult to interpret).

In [11]:
```
topic_model.visualize_barchart(topics = [0,1,2,3,4,5,6,7,8,9,10], n_words = 9)
```


Topic Word Scores
Topic 0
phyrexia
phyrexians
phyrexian
phyrexias
vorinclex
mycosynth
azaxazog
thane
onetime
0 0.01 0.02 0.03 0.04
Topic 1
sword
steel
blade
swords
armor
mokgar
spears
dagger
forge
0 0.005 0.01 0.015
Topic 2
kami
kamigawa
observations
akki
oni
konda
war
takeno
sensei
0 0.01 0.02 0.03
Topic 3
goblins
goblin
demise
rivaled
ridden
innovations
military
sacrificial
bug
0 0.02 0.04
Topic 4
dragons
dragon
caustic
digest
dragontooth
procure
trusted
scale
consuming
0 0.02 0.04
Topic 5
sarpadian
empires
vol
orcs
iii
thallids
doubleedged
orcish
iv
0 0.02 0.04
Topic 6
werewolves
wolf
werewolf
wolves
howl
ulvenwald
wolfir
werewolfs
gatstaf
0 0.01 0.02 0.03 0.04
Topic 7
hunters
hunt
thrashes
hunting
lacerators
prints
companions
boar
viashino
0 0.005 0.01 0.015 0.02
Topic 8
goblin
goblins
ib
halfheart
cabin
tactician
raider
squee
jula
0 0.02 0.04
Topic 9
necromancer
limdl
leshrac
barons
recruitment
liege
necromancers
efreeti
summoners
0 0.02 0.04 0.06 0.08
Topic 10
wildspeaker
garruk
iedo
selfish
magan
skerk
fisher
geor
butcher
0 0.02 0.04

I'm not too familiar with these cards, but through Google searches of the top few words, I was able to come up with what I think are good topic names. I have added supporting links as well.

- Topic 0 - Based on the top words (which show up in 'Phyrexia creature' cards in Google searches), this topic seems to capture the set 'New Phyrexia'.
- Topic 1 - Sword of Sinew and Steel (https://www.cardkingdom.com/mtg/modern-horizons/sword-of-sinew-and-steel)
- Topic 2 - Champions of Kamigawa (https://mtg.wtf/set/chk?page=7)
- Topic 3 - Beetleback Chief (https://gatherer.wizards.com/pages/card/Details.aspx?multiverseid=386305)
- Topic 4 - Noxious Dragon (https://gatherer.wizards.com/pages/card/details.aspx?multiverseid=391888)
- Topic 5 - Sarpadian Empires (https://mtg.fandom.com/wiki/Sarpadian_Empires)
- Topic 6 - Werewolf (https://mtg.fandom.com/wiki/Werewolf)
- Topic 7 - Vampire Lacerator (https://gatherer.wizards.com/pages/card/details.aspx?multiverseid=192225)
- Topic 8 - Squee (Squee was a **goblin cabin-hand** on the Skyship Weatherlight - https://mtg.fandom.com/wiki/Squee)
- Topic 9 - Necromancy (https://www.moxfield.com/decks/rlvIQMx1zUCT6smgX4GpOw)
- Topic 10 - Garruk Wildspeaker (https://gatherer.wizards.com/pages/card/details.aspx?multiverseid=140205)

In [12]:
```
# add topic name
freq_topics_11 = freq_topics.iloc[0:11, :]

freq_topics_11['Topic Name'] = ['New Phyrexia',
                                 'Sword of Sinew and Steel',
                                 'Champions of Kamigawa',
                                'Beetleback Chief',
                                'Noxious Dragon',
                                'Sarpadian Empires',
                                'Werewolf',
                                'Vampire Lacerator',
                                'Squee',
                                'Necromancy',
                                'Garruk Wildspeaker']
```

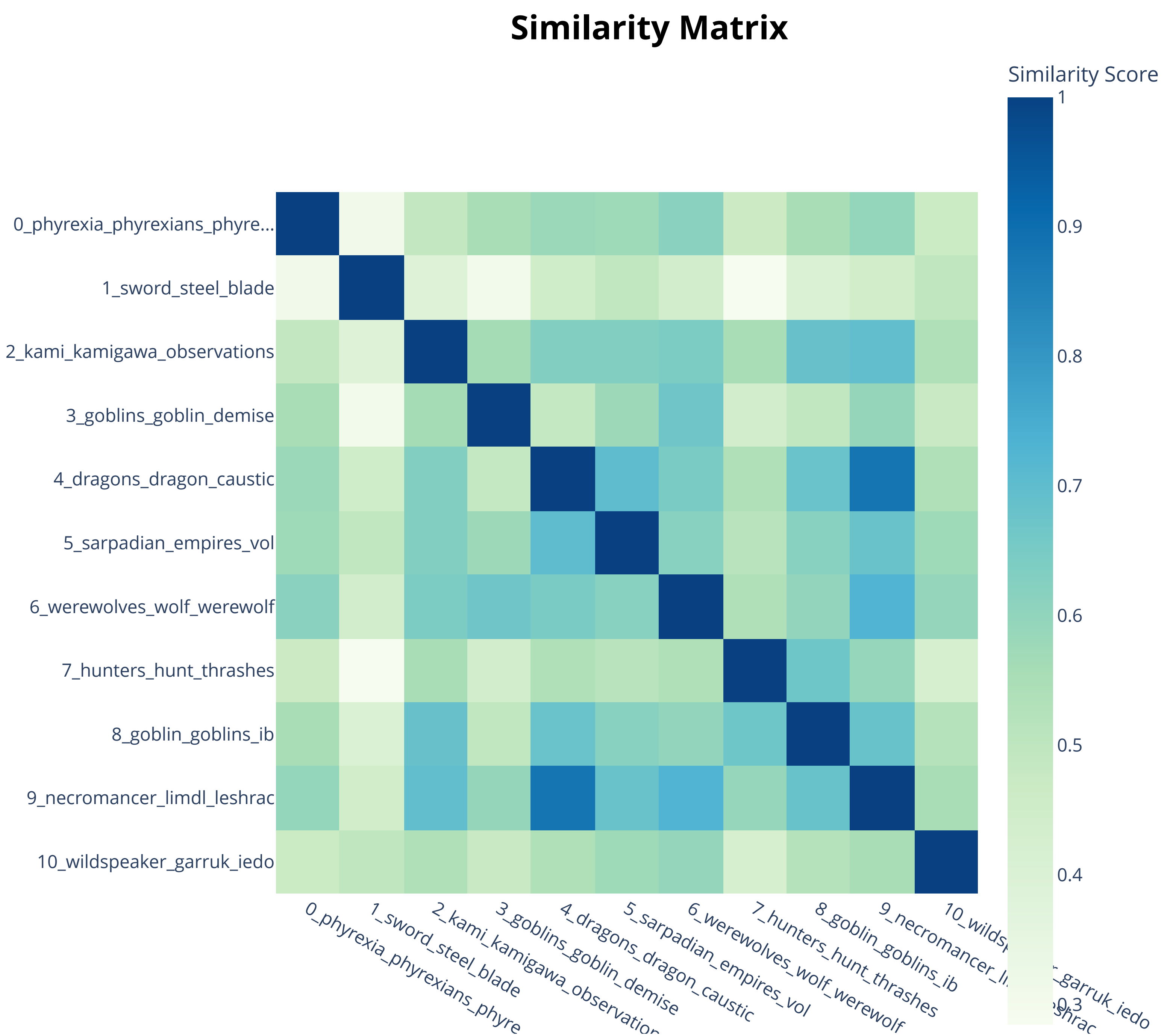
```
# view
freq_topics_11
```

Out[12]:

	Topic	Count	Name	Topic Name
1	0	211	0_phyrexia_phyrexians_phyrexian_phyrexias	New Phyrexia
2	1	205	1_sword_steel_blade_swords	Sword of Sinew and Steel
3	2	203	2_kami_kamigawa_observations_akki	Champions of Kamigawa
4	3	182	3_goblins_goblin_demise_rivaled	Beetleback Chief
5	4	125	4_dragons_dragon_caustic_digest	Noxious Dragon
6	5	125	5_sarpadian_empires_vol_orcs	Sarpadian Empires
7	6	120	6_werewolves_wolf_werewolf_wolves	Werewolf
8	7	119	7_hunters_hunt_thrashes_hunting	Vampire Lacerator
9	8	114	8_goblin_goblins_ib_halfheart	Squee
10	9	98	9_necromancer_limdl_leshrac_barons	Necromancy
11	10	95	10_wildspeaker_garruk_iedo_selfish	Garruk Wildspeaker

In [13]:

```
topic_model.visualize_heatmap(topics = [0,1,2,3,4,5,6,7,8,9,10])
```


Similarity Matrix
Similarity Score
1
0.9
0.8
0.7
0.6
0.5
0.4
0.3
0_phyrexia_phyrexians_phyre...
1_sword_steel_blade
2_kami_kamigawa_observations
3_goblins_goblin_demise
4_dragons_dragon_caustic
5_sarpadian_empires_vol
6_werewolves_wolf_werewolf
7_hunters_hunt_thrashes
8_goblin_goblins_ib
9_necromancer_limdl_leshrac
10_wildspeaker_garruk_iedo

A heatmap shows the similarity between topics (based on the cosine similarity matrix between topic embeddings). Looking at the heatmap above, we can see that topic 9 (Necromancy) is similar to topic 4 (Noxious Dragon).

Once you have names, create a Dynamic Topic Model by following their documentation. Use the release_date column as timestamps.

In [14]:
```
df2 = df.dropna(how = 'any', subset = ['flavor_text'])

# check if dataframe has any missing values in the release_date column
df2.isnull().sum()
```

Out[14]:
```
color_identity            0
colors                    0
converted_mana_cost       0
edhrec_rank             228
keywords              19162
mana_cost              1731
name                      0
number                    0
power                 14357
rarity                 1137
subtypes                  0
supertypes                0
text                      0
toughness             14336
types                     0
flavor_text               0
life                  29603
code                      0
release_date              0
block                 12819
dtype: int64
```

In [15]:
```
# store release_date column as list
timestamps = df2.release_date.to_list()
```

```
# check length
len(timestamps)
```

Out[15]: 29635

In [16]:
```
# store flavor_text data as list
flavor_text_list = df2.flavor_text.tolist()

# check length
len(flavor_text_list)
```

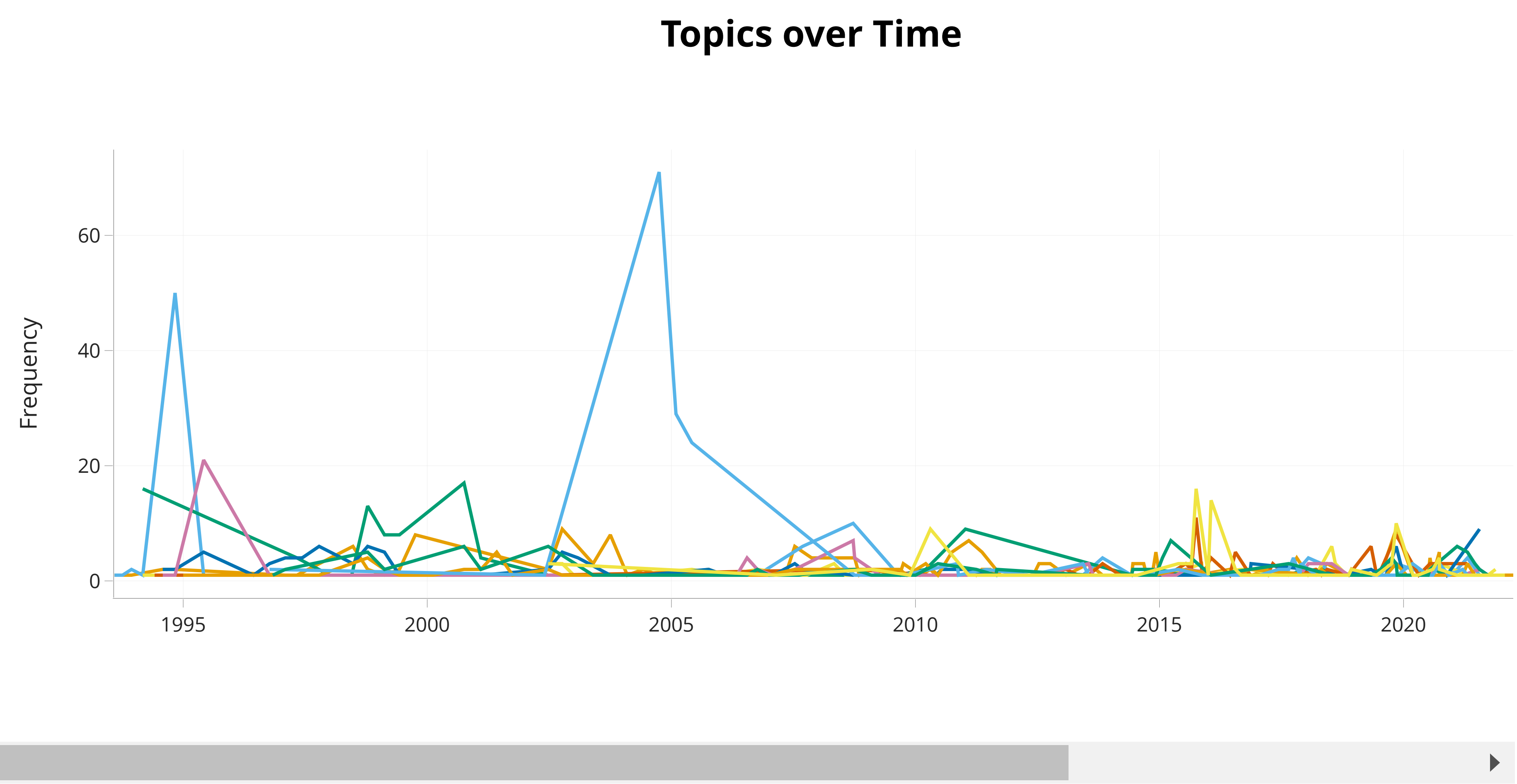
Out[16]: 29635

In [17]:
```
# fit model again
topics, probs = topic_model.fit_transform(flavor_text_list)

# check length of topics
len(topics)
```

Out[17]: 29635

In [18]:
```
# generate the topic representations at each timestamp for each topic
topics_over_time = topic_model.topics_over_time(flavor_text_list, topics, timestamps)
```

In [19]:
```
topic_model.visualize_topics_over_time(topics_over_time, topics = [0,1,2,3,4,5,6,7,8,9,10])
```



`Champions of Kamigawa` was released in October 2004 (which explains the spike around 2005).

Part 2 Supervised Classification

Using only the text and flavor_text data, predict the color identity of cards:

Follow the sklearn documentation covered in class on text data and Pipelines to create a classifier that predicts which of the colors a card is identified as. You will need to preprocess the target color_identity labels depending on the task:

- Source code for pipelines

- in multiclass.py, again load data and train a Pipeline that preprocesses the data and trains a multiclass classifier (LinearSVC), and saves the model pickel output once trained. target labels with more than one color should be unlabeled!
- in multilabel.py, do the same, but with a multilabel model (e.g. here). You should now use the original color_identity data as-is, with special attention to the multi-color cards.

- in dvc.yaml, add these as stages to take the data and scripts as input, with the trained/saved models as output.

- in your notebook:
  - **Describe: preprocessing steps (the tokenization done, the ngram_range, etc.), and why.**
  - **load both models and plot the confusion matrix for each model (see here for the multilabel-specific version)**
  - **Describe: what are the models succeeding at? Where are they struggling? How do you propose addressing these weaknesses next time?**

Multiclass Classifier

In [21]:
```
# check missing values
df.isnull().sum()
```

Out[21]:
```
color_identity             0
colors                     0
converted_mana_cost        0
edhrec_rank             4123
keywords               33852
mana_cost               7043
name                       0
number                     0
power                  30179
rarity                  4225
subtypes                   0
supertypes                 0
text                       0
toughness              30118
types                      0
flavor_text            26731
life                   56247
code                       0
release_date               0
block                  27689
dtype: int64
```

`color_identity` and `text` don't have any missing values so only missing values from the `flavor_text` variable need to be removed.

In [22]:
```
# remove rows where target (color_identity) or predictors (flavor_text and text) have missing values
df2 = df.dropna(how = 'any',
                subset = ['flavor_text'])

# check
df2.isnull().sum()
```

Out[22]:
```
color_identity            0
colors                    0
converted_mana_cost       0
edhrec_rank             228
keywords              19162
mana_cost              1731
name                      0
number                    0
power                 14357
rarity                 1137
subtypes                  0
supertypes                0
text                      0
toughness             14336
types                     0
flavor_text               0
life                  29603
code                      0
release_date              0
block                 12819
dtype: int64
```

For x, combine text and flavor text data

In [23]:
```
df2['combined_text'] = df['text'] + ' ' + df['flavor_text']

# view
df2.head(2)
```

Out[23]:

	color_identity	colors	converted_mana_cost	edhrec_rank	keywords	mana_cost	name	number	power	rarity	...	supertypes	text	tougl
1	[W]	[W]	5.0	14430.0	[Flying]	[4, W]	Angel of Mercy	2	3.0	uncommon	...	[]	Flying When Angel of Mercy enters the battlefi...	
3	[W]	[W]	4.0	14972.0	None	[3, W]	Ballista Squad	8	2.0	uncommon	...	[]	{X}{W}, {T}: Ballista Squad deals X damage to ...	

2 rows × 21 columns

For y, encode target variable (`color_identity`)

Target labels with more than one color should be unlabeled!

To "unlabel" data, I will replace the label with -1.
Where there are no values, I will replace the label to null

In [24]:
```
# store color_identity values as a list
color_identity_values = list(df2.color_identity.values)

# create empty list to store results
color_identity_multiclass = []

# iterate through list, and unlabel target labels with more than one color
for i in color_identity_values:
    if len(i) == 1:
        color_identity_multiclass.append(i[0])
    elif len(i) < 1:
        color_identity_multiclass.append(0) # storing missing values as 0
```

```
    else:
        color_identity_multiclass.append(-1) # unlabeling target labels with more than one color

# check length
len(color_identity_multiclass)
```

Out[24]: 29635

In [25]:
```
# check target labels
set(color_identity_multiclass)
```

Out[25]: {-1, 0, 'B', 'G', 'R', 'U', 'W'}

In [26]:
```
### encode target labels (I will do this manually instead of using LabelEncoder())

# store empty list to append to later
encoded_target_multiclass = []

for i in color_identity_multiclass:
    if i == 'W':
        encoded_target_multiclass.append(1)
    elif i == 'U':
        encoded_target_multiclass.append(2)
    elif i == 'R':
        encoded_target_multiclass.append(3)
    elif i == 'G':
        encoded_target_multiclass.append(4)
    elif i == 'B':
        encoded_target_multiclass.append(5)
    elif i == -1:
        encoded_target_multiclass.append(i)
    else:
        encoded_target_multiclass.append(i)

# check length
len(encoded_target_multiclass)
```

Out[26]: 29635

In [27]:
```
# check labels
set(encoded_target_multiclass)
```

Out[27]: {-1, 0, 1, 2, 3, 4, 5}

In [28]:
```
# add encoded labels to dataframe as a new column
df2['multiclass'] = encoded_target_multiclass

# view
df2.head(2)
```

Out[28]:

	color_identity	colors	converted_mana_cost	edhrec_rank	keywords	mana_cost	name	number	power	rarity	...	text	toughness	ty
1	[W]	[W]	5.0	14430.0	[Flying]	[4, W]	Angel of Mercy	2	3.0	uncommon	...	Flying When Angel of Mercy enters the battlefi...	3.0	[Creat
3	[W]	[W]	4.0	14972.0	None	[3, W]	Ballista Squad	8	2.0	uncommon	...	{X}{W}, {T}: Ballista Squad deals X damage to ...	2.0	[Creat

2 rows × 22 columns

Split data into training and test sets

In [29]:
```
# store target and predictor
y = df2[['multiclass']]
X = df2[['combined_text']]

# split data into training and test sets
train_X, test_X, train_y, test_y = train_test_split(X, y , test_size = .25, random_state = 123)
```

In [30]:
```
# check training and test data shapes
print(train_X.shape[0]/df2.shape[0])
```

```
print(test_X.shape[0]/df2.shape[0])
```

```
0.7499915640290198
0.25000843597098027
```

Training Data

In [31]:
```
# store training data as a list
training_X = train_X.combined_text.tolist()

# check length
len(training_X)
```

Out[31]:
```
22226
```

In [32]:
```
# check train_y length
len(train_y)
```

Out[32]:
```
22226
```

In [33]:
```
# store training target as numpy array
training_target = train_y.multiclass.values

# check length
len(training_target)
```

Out[33]:
```
22226
```

Test Data

In [34]:
```
# store test data as a list
test_x = test_X.combined_text.tolist()

# check length
len(test_x)
```

Out[34]:
```
7409
```

In [35]:
```
# check test_y length
len(test_y)
```

Out[35]: 7409

In [36]:
```
# store test target as numpy array
test_target = test_y.multiclass.values

# check length
len(test_target)
```

Out[36]: 7409

Preprocessing Steps:

Pre-processing text using CountVectorizer():

- removing English stop words in order to remove the 'low-level' information in the text and focus more on the important information.
- converting all words to lowercase - assumption is that the meaning and significance of a lowercase word is the same as when that word is in uppercase or capitalized. This will help remove noise.
- ngram_range set to 1,2 i.e. capturing both unigrams and bigrams since Magic Card texts often have names/terms that are bigrams e.g. Soul Warden and Beetleback Chief.
- min_df set to 5 i.e. rare words that appear in less than 5 documents will be ignored.
- max_df set to 0.9 i.e. words that appear in more than 90% of the documents will be ignored since they are not adding much to a specific document.

Using TfidfTransformer():

- Term frequencies calculated to overcome the discrepancies with using occurence count for differently sized documents.
- Downscaled weights for words that occur in many documents and therefore do not add a lot of information than those that occur in a smaller share of the corpus (tf-idf)

In [37]:
```
# load multiclass model
file_to_read = open("multiclass_classifier.pickle", "rb")
multiclass_classifier = pickle.load(file_to_read)
file_to_read.close()

# view
print(multiclass_classifier)
```

```
Pipeline(steps=[('vect',
                 CountVectorizer(max_df=0.9, min_df=5, ngram_range=(1, 2),
                                 stop_words='english')),
                ('tfidf', TfidfTransformer()), ('clf', LinearSVC())])
```

In [38]:
```
predicted = multiclass_classifier.predict(test_x)
np.mean(predicted == test_target)
```

Out[38]:
```
0.8501822108246727
```

We achieved 85% accuracy using Linear SVC.

In [39]:
```
# plot confusion matrix
multilabel_confusion_matrix(test_target, predicted, labels = [1,2,3,4,5])
```

Out[39]:
```
array([[[6023,  208],
        [ 171, 1007]],

       [[6198,  168],
        [ 101,  942]],

       [[6085,  153],
        [ 124, 1047]],

       [[6096,  168],
        [ 162,  983]],

       [[6129,  162],
        [ 167,  951]]], dtype=int64)
```

This is how we can interpret the confusion matrix values: 6023 of the observations with the label 1 (i.e. color White) were predicted correctly by the model, whereas 1007 observations that did not have the label 1 were predicted correctly by the model. 208 records that did not have the label 1 were wrongy predicted as having the label 1, while 171 records that did have the label 1 were wrongly predicted as not having the label 1.

F1 Score

In [40]:
```
# Opening JSON file
f = open('metrics.json')

# returns JSON object as
# a dictionary
```

```
data = json.load(f)

# print
data
```

Out[40]:

```
{'-1': {'precision': 0.8106448311156602,
  'recall': 0.7492904446546831,
  'f1-score': 0.7787610619469026,
  'support': 1057},
 '0': {'precision': 0.8973561430793157,
  'recall': 0.8278335724533716,
  'f1-score': 0.8611940298507462,
  'support': 697},
 '1': {'precision': 0.8288065843621399,
  'recall': 0.8548387096774194,
  'f1-score': 0.8416213957375679,
  'support': 1178},
 '2': {'precision': 0.8486486486486486,
  'recall': 0.9031639501438159,
  'f1-score': 0.8750580585229911,
  'support': 1043},
 '3': {'precision': 0.8725,
  'recall': 0.8941076003415884,
  'f1-score': 0.883171657528469,
  'support': 1171},
 '4': {'precision': 0.8540399652476107,
  'recall': 0.8585152838427947,
  'f1-score': 0.8562717770034843,
  'support': 1145},
 '5': {'precision': 0.8544474393530997,
  'recall': 0.8506261180679785,
  'f1-score': 0.8525324966382788,
  'support': 1118},
 'accuracy': 0.8501822108246727,
 'macro avg': {'precision': 0.852349087400925,
  'recall': 0.848339382740236,
  'f1-score': 0.8498014967469201,
  'support': 7409},
 'weighted avg': {'precision': 0.8501321382831634,
  'recall': 0.8501822108246727,
  'f1-score': 0.8496794125224192,
  'support': 7409}}
```

In [41]:

```
# Closing file
```

```
f.close()
```

In [42]:
```
# store scores as a dataframe
metrics = pd.DataFrame(metrics.classification_report(test_target, predicted, output_dict = True))
print(metrics)
```

```
                    -1           0            1            2            3  \
precision     0.810645    0.897356     0.828807     0.848649     0.872500   
recall        0.749290    0.827834     0.854839     0.903164     0.894108   
f1-score      0.778761    0.861194     0.841621     0.875058     0.883172   
support    1057.000000  697.000000  1178.000000  1043.000000  1171.000000   

                     4            5  accuracy    macro avg  weighted avg  
precision     0.854040     0.854447  0.850182     0.852349      0.850132  
recall        0.858515     0.850626  0.850182     0.848339      0.850182  
f1-score      0.856272     0.852532  0.850182     0.849801      0.849679  
support    1145.000000  1118.000000  0.850182  7409.000000   7409.000000  
```

The macro-averaged F1-score is computed as a simple arithmetic mean of the per-class F1-scores.

When averaging the macro-F1, we gave equal weights to each class. We don’t have to do that: in weighted-average F1-score, we weight the F1-score of each class by the number of samples from that class.

Multilabel Classifier

In [43]:
```
# check missing values
df.isnull().sum()
```

Out[43]:
```
color_identity            0
colors                    0
converted_mana_cost       0
edhrec_rank            4123
keywords              33852
mana_cost              7043
name                      0
number                    0
power                 30179
rarity                 4225
subtypes                  0
supertypes                0
text                      0
toughness             30118
types                     0
flavor_text           26731
life                  56247
code                      0
release_date              0
block                 27689
dtype: int64
```

`color_identity` and `text` don't have any missing values so only missing values from the `flavor_text` variable need to be removed.

In [44]:
```
# remove rows where target (color_identity) or predictors (flavor_text and text) have missing values
df2 = df.dropna(how = 'any',
                subset = ['flavor_text'])

# check
df2.isnull().sum()
```

Out[44]:
```
color_identity            0
colors                    0
converted_mana_cost       0
edhrec_rank             228
keywords              19162
mana_cost              1731
name                      0
number                    0
power                 14357
rarity                 1137
subtypes                  0
supertypes                0
text                      0
toughness             14336
types                     0
flavor_text               0
life                  29603
code                      0
release_date              0
block                 12819
dtype: int64
```

For x, combine text and flavor text data

In [45]:
```
df2['combined_text'] = df['text'] + ' ' + df['flavor_text']

# view
df2.head(2)
```

Out[45]:

	color_identity	colors	converted_mana_cost	edhrec_rank	keywords	mana_cost	name	number	power	rarity	...	supertypes	text	tougl
1	[W]	[W]	5.0	14430.0	[Flying]	[4, W]	Angel of Mercy	2	3.0	uncommon	...	[]	Flying When Angel of Mercy enters the battlefi...	
3	[W]	[W]	4.0	14972.0	None	[3, W]	Ballista Squad	8	2.0	uncommon	...	[]	{X}{W}, {T}: Ballista Squad deals X damage to ...	

2 rows × 21 columns

For y, use the (`color_identity`) column as is

Guidance obtained from: https://scikit-learn.org/stable/modules/preprocessing_targets.html#preprocessing-targets

In [46]:
```
# store color_identity values as a list
color_identity_values = list(df2.color_identity.values)

# create label binary indicator array - target
color_identity_multilabels = MultiLabelBinarizer().fit_transform(color_identity_values)
```

In [47]:
```
# store target and predictor
y = color_identity_multilabels
X = df2[['combined_text']]

# split data into training and test sets
train_X, test_X, train_y, test_y = train_test_split(X, y , test_size = .25, random_state = 123)
```

In [48]:
```
# check training and test data shapes
print(train_X.shape[0]/df2.shape[0])
```

```
print(test_X.shape[0]/df2.shape[0])
```

```
0.7499915640290198
0.25000843597098027
```

Training Data

In [49]:
```
# store training data as a list
training_X = train_X.combined_text.tolist()

# check length
len(training_X)
```

Out[49]:
```
22226
```

In [50]:
```
# check train_y length
len(train_y)
```

Out[50]:
```
22226
```

In [51]:
```
# store training target as numpy array
training_target = train_y

# check length
len(training_target)
```

Out[51]:
```
22226
```

Test Data

In [52]:
```
# store test data as a list
test_x = test_X.combined_text.tolist()

# check length
len(test_x)
```

Out[52]:
```
7409
```

In [53]:
```
# check test_y length
len(test_y)
```

Out[53]: 7409

In [54]:
```
# store test target as numpy array
test_target = test_y

# check length
len(test_target)
```

Out[54]: 7409

Preprocessing Steps:

Pre-processing text using CountVectorizer():

- removing English stop words in order to remove the 'low-level' information in the text and focus more on the important information.
- converting all words to lowercase - assumption is that the meaning and significance of a lowercase word is the same as when that word is in uppercase or capitalized. This will help remove noise.
- ngram_range set to 1,2 i.e. capturing both unigrams and bigrams since Magic Card texts often have names/terms that are bigrams e.g. Soul Warden and Beetleback Chief.
- min_df set to 5 i.e. rare words that appear in less than 5 documents will be ignored.
- max_df set to 0.9 i.e. words that appear in more than 90% of the documents will be ignored since they are not adding much to a specific document.

Using TfidfTransformer():

- Term frequencies calculated to overcome the discrepancies with using occurence count for differently sized documents.
- Downscaled weights for words that occur in many documents and therefore do not add a lot of information than those that occur in a smaller share of the corpus (tf-idf)

In [55]:
```
# load multilabel model
file_to_read = open("multilabel_classifier.pickle", "rb")
multilabel_classifier = pickle.load(file_to_read)
file_to_read.close()

# view
print(multilabel_classifier)
```

```
Pipeline(steps=[('vect',
                 CountVectorizer(max_df=0.9, min_df=5, ngram_range=(1, 2),
                                 stop_words='english')),
                ('tfidf', TfidfTransformer()),
                ('clf', OneVsRestClassifier(estimator=SVC(kernel='linear')))])
```

In [56]:
```
predicted = multilabel_classifier.predict(test_x)
np.mean(predicted == test_target)
```

Out[56]:
```
0.9330004049129437
```

We achieved 93% accuracy using OneVsRestClassifier.

In [57]:
```
# plot confusion matrix
multilabel_confusion_matrix(test_target, predicted)
```

Out[57]:
```
array([[[5719,  119],
        [ 359, 1212]],

       [[5646,  117],
        [ 402, 1244]],

       [[5682,  127],
        [ 329, 1271]],

       [[5792,  134],
        [ 314, 1169]],

       [[5565,  170],
        [ 411, 1263]]], dtype=int64)
```

Part 3

Part 3: Regression?

Can we predict the EDHREC "rank" of the card using the data we have available?

- Like above, add a script and dvc stage to create and train your model
- in the notebook, aside from your descriptions, plot the predicted vs. actual rank, with a 45-deg line showing what "perfect prediction" should look like.

- This is a freeform part, so think about the big picture and keep track of your decisions:
  - what model did you choose? Why?
  - What data did you use from the original dataset? How did you proprocess it?
  - Can we see the importance of those features? e.g. logistic weights?
- How did you do? What would you like to try if you had more time?

For this part, I wanted to try using some categorical variables that I thought could be important predictors - namely the block i.e. sets with "shared mechanics", and the rarity of cards.

I ran a grid search using K-nearest neighbors, random forest and a decision tree regressor, and found KNN() with 5-nearest neighbors to be the best model.

In [58]:
```
# remove rows where target or predictors have missing values
df2 = df.dropna(how = 'any',
                subset = ['block',
                         'rarity',
                         'edhrec_rank'])
```

```
block
```

In [59]:
```
# get dummies
block_dummies = pd.get_dummies(df2.block)
block_dummies.columns = [c.lower().replace(" ","_") for c in block_dummies.columns]

block_dummies = block_dummies.drop(['alara'],axis=1) # Baseline
block_dummies.head(5)
```

Out[59]:

	amonkhet	arena_league	battle_for_zendikar	commander	conspiracy	core_set	friday_night_magic	guilds_of_ravnica	ice_age	innistrad	...	ravnica
0	0	0	0	0	0	1	0	0	0	0	...	0
1	0	0	0	0	0	1	0	0	0	0	...	0
2	0	0	0	0	0	1	0	0	0	0	...	0
3	0	0	0	0	0	1	0	0	0	0	...	0
4	0	0	0	0	0	1	0	0	0	0	...	0

5 rows × 34 columns

In [60]:
```
df2 = pd.concat([df2.drop(['block'],axis=1),block_dummies],axis=1)
df2.head()
```

Out[60]:

	color_identity	colors	converted_mana_cost	edhrec_rank	keywords	mana_cost	name	number	power	rarity	...	ravnica	return_to_ravr
0	[W]	[W]	7.0	16916.0	[First strike]	[5, W, W]	Ancestor's Chosen	1	4.0	uncommon	...	0	
1	[W]	[W]	5.0	14430.0	[Flying]	[4, W]	Angel of Mercy	2	3.0	uncommon	...	0	
2	[W]	[W]	4.0	13098.0	[Flying]	[3, W]	Aven Cloudchaser	7	2.0	common	...	0	
3	[W]	[W]	4.0	14972.0	None	[3, W]	Ballista Squad	8	2.0	uncommon	...	0	
4	[W]	[W]	1.0	4980.0	None	[W]	Bandage	9	NaN	common	...	0	

5 rows × 53 columns

rarity

In [61]:
```
# get dummies
rarity_dummies = pd.get_dummies(df2.rarity)
rarity_dummies.columns = [c.lower().replace(" ","_") for c in rarity_dummies.columns]
```

```
rarity_dummies = rarity_dummies.drop(['common'],axis=1) # Baseline
rarity_dummies.head(5)
```

Out[61]:

	rare	uncommon
0	0	1
1	0	1
2	0	0
3	0	1
4	0	0

In [62]:
```
df2 = pd.concat([df2.drop(['rarity'],axis=1),rarity_dummies],axis=1)
df2.head()
```

Out[62]:

	color_identity	colors	converted_mana_cost	edhrec_rank	keywords	mana_cost	name	number	power	subtypes	...	scars_of_mirrodin	shadc
0	[W]	[W]	7.0	16916.0	[First strike]	[5, W, W]	Ancestor's Chosen	1	4.0	[Human, Cleric]	...	0	
1	[W]	[W]	5.0	14430.0	[Flying]	[4, W]	Angel of Mercy	2	3.0	[Angel]	...	0	
2	[W]	[W]	4.0	13098.0	[Flying]	[3, W]	Aven Cloudchaser	7	2.0	[Bird, Soldier]	...	0	
3	[W]	[W]	4.0	14972.0	None	[3, W]	Ballista Squad	8	2.0	[Human, Rebel]	...	0	
4	[W]	[W]	1.0	4980.0	None	[W]	Bandage	9	NaN	[]	...	0	

5 rows × 54 columns

In [63]:
```
# store target and predictor
y = df2[['edhrec_rank']]
X = df2[['amonkhet', 'arena_league',
       'battle_for_zendikar', 'commander', 'conspiracy', 'core_set',
       'friday_night_magic', 'guilds_of_ravnica', 'ice_age', 'innistrad',
```

```
        'innistrad:_double_feature', 'invasion', 'ixalan', 'judge_gift_cards',
        'kaladesh', 'kamigawa', 'khans_of_tarkir', 'lorwyn',
        'magic_player_rewards', 'masques', 'mirage', 'mirrodin', 'odyssey',
        'onslaught', 'ravnica', 'return_to_ravnica', 'scars_of_mirrodin',
        'shadowmoor', 'shadows_over_innistrad', 'tempest', 'theros',
        'time_spiral', 'urza', 'zendikar', 'rare', 'uncommon']]

# split data into training and test sets
train_X, test_X, train_y, test_y = train_test_split(X, y , test_size = .25, random_state = 123)
```

In [64]:
```
# Load model
file_to_read = open("best_mod.pickle", "rb")
best_mod = pickle.load(file_to_read)
file_to_read.close()

# view
print(best_mod)
```

```
Pipeline(steps=[('model', KNeighborsRegressor())])
```

And Run

In [65]:
```
best_mod.fit(train_X,train_y)
```

Out[65]:
```
Pipeline(steps=[('model', KNeighborsRegressor())])
```

In [66]:
```
predicted = best_mod.predict(test_X)
np.mean(predicted == test_y)
```

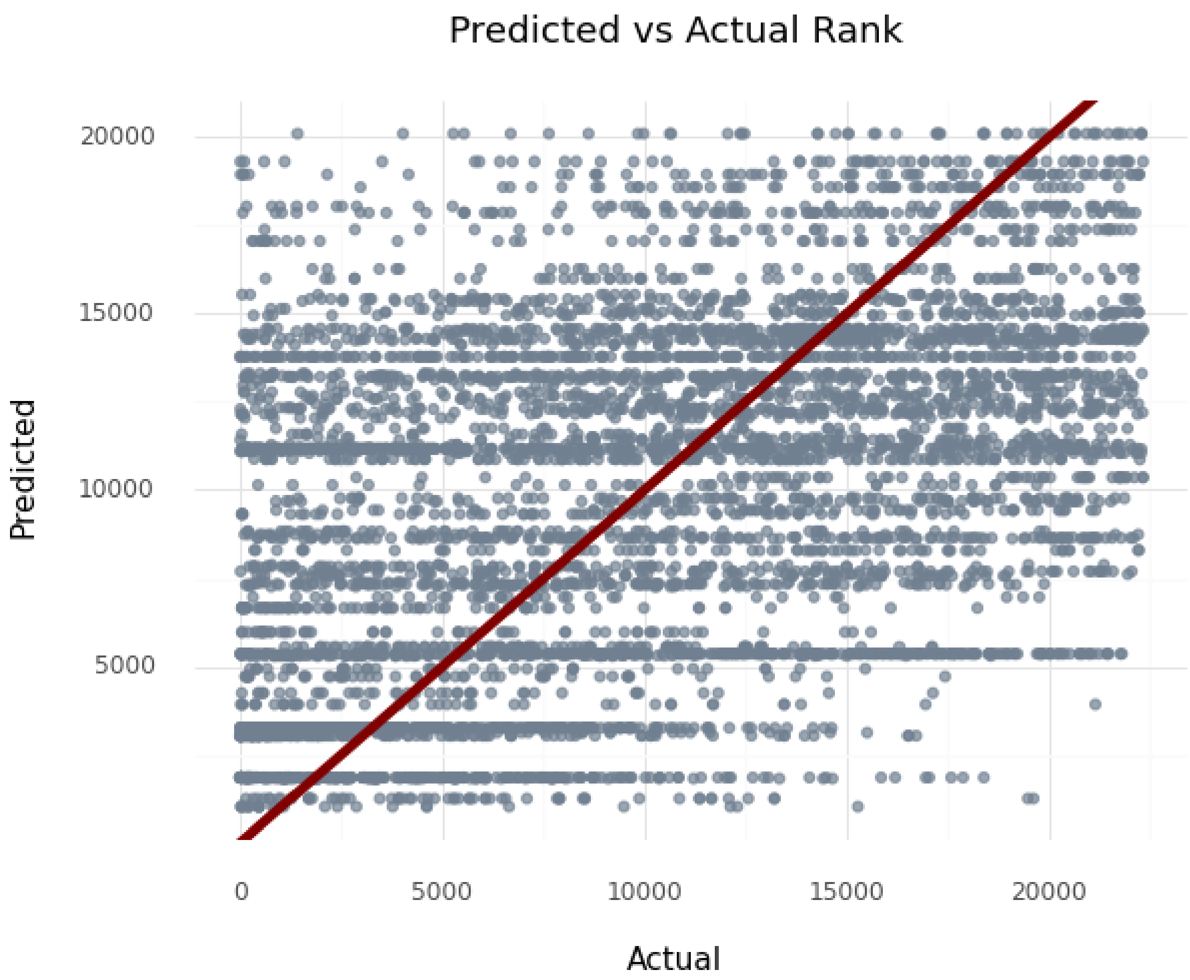
Out[66]:
```
edhrec_rank    0.0
dtype: float64
```

In [67]:
```
# store test_y
df_plot = test_y.copy()

# create empty list
predictions = []

# iterate
for i in predicted:
    predictions.append(i[0])
```

```
# store list as dataframe column
df_plot['predicted'] = predictions
```

In [68]:

```
# plot

(ggplot(data = df_plot,
        mapping = aes(x = 'edhrec_rank', y = 'predicted')) +
 geom_point(color = 'slategray', alpha = 0.7) +
 geom_abline(intercept = 0, slope = 1, size = 2, color = 'maroon') +
 theme_minimal() +
 labs(title = 'Predicted vs Actual Rank\n',
     y = 'Predicted\n',
     x = '\nActual')
)
```



Out[68]: <ggplot: (130277217717)>

This isn't a good plot since the dots are scattered everywhere instead of being close to the line (i.e. predictions being close to the actual values.

KNN() with 5 nearest neighbors was identified as the best model when I did a grid search. However, when I loaded the model in the notebook, it did not have the number of neighbors specified and I was unsure how to add it or how to save the model in the .py script such that the number of neighbors also gets saved as a parameter of KNN.

How did I do? Not too great. Definitely a lot of room for improvement. I would like to select more predictors if I have more time, as well as include k=5 in the KNN regressor (update: the default number of neighbors is 5 so even though I didn't specify k, the model ran with k=5).

Part 4

For multiclass, report average and F1

Done above where the multiclass model was run.

Run a new experiment that changes one parameter:

output of `dvc exp diff` copy and pasted from the command line, and formatted to a table:

Path	Metric	exp-ddb8e	workspace	Change
metrics.json	-1.f1-score	0.78642	0.77876	-0.0076563
metrics.json	-1.precision	0.81949	0.81064	-0.0088423
metrics.json	-1.recall	0.75591	0.74929	-0.0066225
metrics.json	0.f1-score	0.8684	0.86119	-0.0072075
metrics.json	0.precision	0.90123	0.89736	-0.0038784
metrics.json	0.recall	0.83788	0.82783	-0.010043
metrics.json	1.f1-score	0.83958	0.84162	0.0020424
metrics.json	1.precision	0.83292	0.82881	-0.004109
metrics.json	1.recall	0.84635	0.85484	0.008489
metrics.json	2.f1-score	0.87825	0.87506	-0.0031947
metrics.json	2.precision	0.85212	0.84865	-0.0034704
metrics.json	2.recall	0.90604	0.90316	-0.0028763
metrics.json	3.f1-score	0.8846	0.88317	-0.0014276
metrics.json	3.precision	0.86964	0.8725	0.002863
metrics.json	3.recall	0.90009	0.89411	-0.0059778

Path	Metric	exp-ddb8e	workspace	Change
metrics.json	4.f1-score	0.85777	0.85627	-0.0014944
metrics.json	4.precision	0.85702	0.85404	-0.0029783
metrics.json	5.f1-score	0.85931	0.85253	-0.0067797
metrics.json	5.precision	0.85816	0.85445	-0.0037149
metrics.json	5.recall	0.86047	0.85063	-0.009839
metrics.json	accuracy	0.85356	0.85018	-0.0033743
metrics.json	macro avg.f1-score	0.85348	0.8498	-0.0036739
metrics.json	macro avg.precision	0.8558	0.85235	-0.0034472
metrics.json	macro avg.recall	0.85218	0.84834	-0.0038385
metrics.json	weighted avg.f1-score	0.85305	0.84968	-0.0033749
metrics.json	weighted avg.precision	0.85347	0.85013	-0.0033366
metrics.json	weighted avg.recall	0.85356	0.85018	-0.0033743

Path	Param	exp-ddb8e	workspace	Change
params.yaml	preprocessing.ngrams.largest	3	2	-1

Grabbing the weighted average scores from the output above:

	Precision	Recall	F1-Score
ngrams.largest = 2	0.85013	0.85018	0.84968
ngrams.largest = 3	0.85347	0.85356	0.85305

There was only a very slight improvement in performance when the ngram range was changed from (1,2) to (1,3), based on the slightly higher scores.