# Homework II: Goals and Approaches

PPOL628 Text as Data: Computational Linguistics

Alexander Adams

2022-Apr-26

[1]:
```
#Run dvc
!dvc pull
```

```
Everything is up to date.
```

[2]:
```
from bertopic import BERTopic
import pandas as pd
```

[3]:
```
#Import data
mtg = (pd.read_feather('../../../data/mtg.feather'))
mtg = mtg.dropna(subset = ['flavor_text']).reset_index(drop=True)
# .head()[['name','text', 'mana_cost', 'flavor_text','release_date',␣
 ↪'edhrec_rank']]
mtg.shape
```

[3]:
```
(29635, 20)
```

## 1 Part I: Unsupervised Exploration

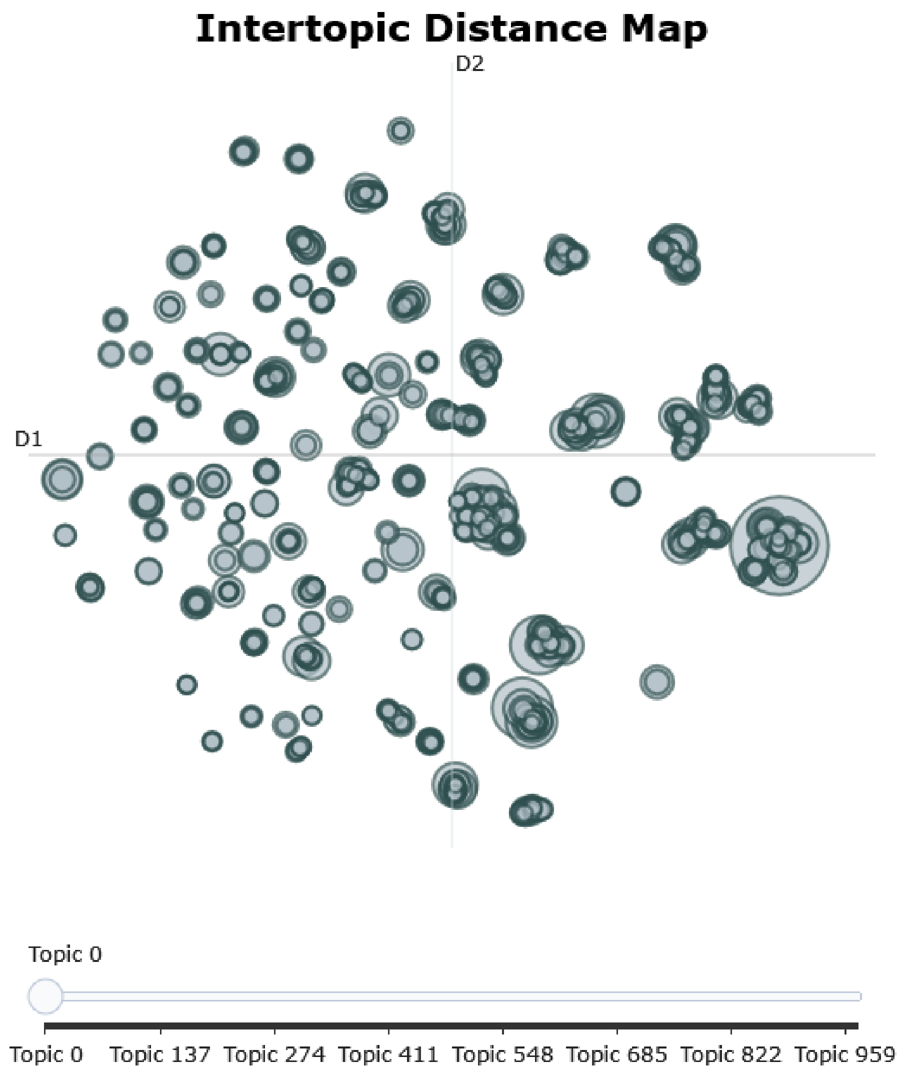
[4]:
```
#Load in topic model generated by topic_model.py
topic_model = BERTopic.load('hw2_bert_model')
```

[5]:
```
topics_list = topic_model.get_topics()
len(topic_model.get_topics())
```

[5]:
```
977
```

When the model is fitted on the flavor text and no additional parameters are specified, BERT finds 977 topics in this corpus. Those topics are plotted in two dimensions below:

[6]:
```
topic_model.visualize_topics(topics_list)
```

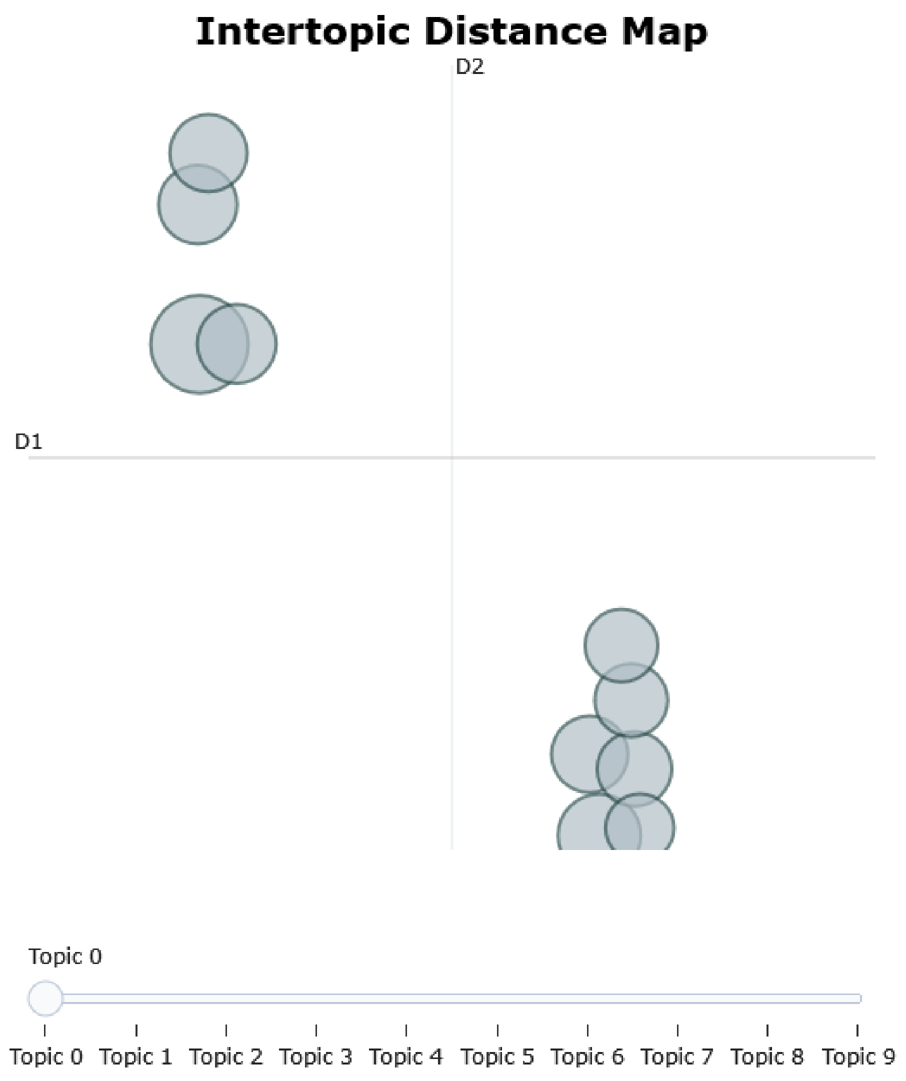


```
[7]: #This is necessary because we're working with an imported model
     #See: https://github.com/MaartenGr/BERTopic/issues/498#issuecomment-1097932864
     probs = topic_model.hdbscan_model.probabilities_
     topics = topic_model._map_predictions(topic_model.hdbscan_model.labels_)
```

If I reduce the number of topics by about 99% (from 977 down to 10), what does the output look like?

```
[8]: #Reduce the number of topics
     new_topics, new_probs = topic_model.reduce_topics(mtg['flavor_text'], topics,␣
      ↪probs, nr_topics = 10)
```

[9]:
```
#Plot the ten topics
topic_model.visualize_topics()
```



I can now display the top words associated with each of the topics. The documentation for BERTopic says that topic #-1 refers to outliers and should be ignored.

[10]:
```
topic_model.get_topic_info()
```

[10]:
```
   Topic  Count                       Name
0     -1  25884            -1_the_of_to_and
1      0    583       0_dragons_the_dragon_of
2      1    418       1_goblins_goblin_the_to
3      2    379    2_vol_empires_sarkhan_the
```

```
4       3    376          3_sun_the_shadow_darkness
5       4    365        4_phyrexia_the_of_mirrodin
6       5    357              5_urza_karn_the_golem
7       6    339  6_weatherlight_gerrard_hanna_the
8       7    326     7_jace_chandra_beleren_nalaar
9       8    323      8_nature_civilization_of_the
10      9    285       9_lightning_the_became_iron
```

Based on these words, it looks like the topics are more-or-less as follows:

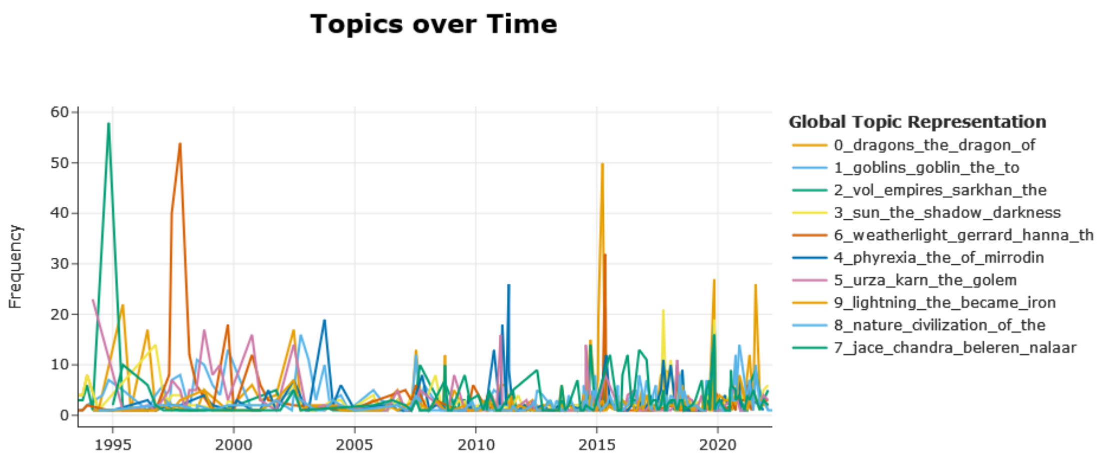
```
1: Dragons
2: Goblins
3: Unknown
4: Light and darkness
5: Unknown
6: Unknown
7: Unknown
8: Unknown
9: Nature and civilization
10: Maybe elemental transmutation? (That sounds like something that would
be present in the game.)
```

Topics 3, 5, 6, 7, and 8 are clearly associated with particular proprietary characters, events, locations, or items in the game; while these topics may be internally coherent within the Magic: The Gathering domain (and may make sense to those more familiar with this subject matter), they are largely inscrutable to non-experts, insofar as these top few words are considered.

Next, I create a dynamic topic model:

```
[11]: dynamic_topics = topic_model.topics_over_time(mtg['flavor_text'],
                                                  new_topics,
                                                  mtg['release_date'])
```

```
[12]: topic_model.visualize_topics_over_time(dynamic_topics,
                                           topics=[0,1,2,3,4,5,6,7,8,9],
                                           width = 950)
```



If the above plot is filtered to just include the topics with recognizable words, certain trends become somewhat visible. For one, the "Dragon" and "Goblin" topics have significant overlap, starting around 2003 when they experience local spikes back to back. The "Dragon" topic is consistently more prominent and has greater local maxima, which could indicate that dragons are more common in the set of available cards, or that they are associated with a large number of other entities.

The "light and darkness" topic experienced a peak around 2017, and again in late 2019. Before these, this topic had only been associated with notable peaks in 2008 and 1996-7.

When all ten topics are included in the plot, we can see that while topics 2 and 6 have local peaks greater than the greatest "Dragon" peak, both of those peaks occurred prior to 2000, and there have been no similar occurrences since then. It is possible that this could indicate a shift away from more propietary concepts in cards and toward more universally-known fantasy tropes and items. Those topics could also be associated with particular collections or decks of cards which have been discontinued or which are part of now-ended series.

---

## 2 Part II: Supervised Classification

### 2.1 Multiclass Classification

First, the multiclass model is loaded, the data is preprocessed to match the format used in the pipeline, and a confusion matrix is generated.

[13]:
```
import joblib
import numpy as np
from sklearn.metrics import (confusion_matrix, multilabel_confusion_matrix,
precision_recall_fscore_support, classification_report)
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MultiLabelBinarizer
```

[14]:
```
#Load the trained multiclass pipeline
multiclass_model = joblib.load('multiclass_pipe.pkl')
```

[15]:
```
#Drop rows with missing values in the variables of interest
mtg = mtg.dropna(subset = ['flavor_text', 'text', 'color_identity']).
 ↪reset_index(drop=True)
```

[16]:
```
#Create numeric labels based on values of colors
#New values: X = multiclass, Z = NaN
mtg['color_label'] = [np.array(['X']) if len(x) > 1 else x for x in␣
 ↪mtg['color_identity']]
mtg['color_label'] = [np.array(['Z']) if len(x) == 0 else x for x in␣
 ↪mtg['color_label']]
mtg['color_label'] = np.concatenate(mtg['color_label'])
```

[17]:
```
#Merge labels into MTG data frame
labels = pd.DataFrame(mtg['color_label'].unique()).reset_index()
#Add one because zero indexed
labels['index'] = labels['index']+1
labels.columns = ['label', 'color_label']
mtg = labels.merge(pd.DataFrame(mtg), how = 'right', on = 'color_label')
```

[19]:
```
#Select labels as targets
y = mtg['label']

#Select text columns as features
X = mtg[['text', 'flavor_text']]

#Training test split 75/25
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

[20]:
```
predicted_mc = multiclass_model.predict(X_test)
```

[21]:
```
multiclass_cm = pd.DataFrame(confusion_matrix(y_test,predicted_mc.round()))
multiclass_cm.index = np.unique(mtg['color_label'])
multiclass_cm.columns = np.unique(mtg['color_label'])
multiclass_cm
```

[21]:
```
      B     G     R     U     W    X    Z
B  1105    12     8     8     8    4    9
G     9  1109     1     4    11    2    9
R     9     5  1023     6     9    4    5
U     8     9     4  1064    10    6    4
W    13     5     5     3  1087    6   13
X     5     5    11     9     5  719    3
Z    13    11    13    14     8    1  995
```

I was somewhat unsure how to tackle this, so I treated "multiclass" as a class itself and labeled it

“X”. I also labeled those cards which were missing a color with “Z”. I know that this was not how this problem was intended to be solved, and that as such I likely achieved different outcomes from what was expected or desired.

Based on the raw confusion matrix, it appears that the Linear Support Vector Classifier was reasonably successful at predicting colors for cards, as well as predicting when a card had no color and when it had multiple colors. However, in order to properly assess the classifier results, it is necessary to calculate precision, recall, and the F-score.

```
[22]: mc_evals = pd.DataFrame()
      for x in np.unique(y_test):
          mc_evals[x] = precision_recall_fscore_support(y_test,
                                                        predicted_mc,
                                                        labels = [x],
                                                        average = 'macro')
```

```
[23]: mc_evals.columns = np.unique(mtg['color_label'])
      mc_evals.index = ['precision','recall','f-score','support']
      mc_evals
```

```
[23]:                   B         G         R         U         W         X  \
      precision  0.950947  0.959343  0.960563  0.960289  0.955185  0.969003
      recall     0.957539  0.968559  0.964185  0.962896  0.960247  0.949802
      f-score    0.954231  0.963929  0.962371  0.961591  0.957709  0.959306
      support         NaN       NaN       NaN       NaN       NaN       NaN

                        Z
      precision  0.958574
      recall     0.943128
      f-score    0.950788
      support         NaN
```

The output above indicates that for all labels in this problem, precision, recall, and f-score were approximately 0.95. There is a high degree of consistency across classes, which suggests that the text and flavor text provided with each card are useful predictors for each class and for all classes. A precision score of 0.95 means that 95% of the instances classified with a particular label actually correspond to that label. A recall score of 0.95 means that 95% of the instances in the dataset with a particular label were classified with that label. The F-score is a measure of accuracy which depends on both precision and recall. A high F-score means that the classifier or model is highly accurate at assigning correct labels and minimizing both false positives and false negatives.

## 2.2 Multilabel Classification

Next, the multilabel model is loaded, and that confusion matrix is generated:

```
[24]: multilabel_model = joblib.load('multilabel_pipe.pkl')
```

```
[25]: for letter in np.unique(np.concatenate(np.array(mtg['color_identity']))):
          mtg["is_"+letter] = [1 if letter in x else 0 for x in mtg['color_identity']]
```

[26]:
```
letters = mtg.columns.str.contains('is_')

mtg['labels'] = mtg[mtg.columns[letters]].values.tolist()
```

[27]:
```
#Select labels as targets
mlb = MultiLabelBinarizer()
y = mlb.fit_transform(mtg['labels'])
#Select text columns as features
X = mtg[['text', 'flavor_text']]

#Training test split 75/25
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)
```

[28]:
```
predicted_ml = multilabel_model.predict(X_test)
```

[29]:
```
multilabel_cm = multilabel_confusion_matrix(y_test,predicted_ml.round())
multilabel_cm
```

[29]:
```
array([[[  18,    1],
        [   0, 7390]],

       [[ 582,  131],
        [   8, 6688]]], dtype=int64)
```

[30]:
```
precision_recall_fscore_support(y_test, predicted_ml, average = 'macro')
```

[30]:
```
(0.9903268361506086, 0.9994026284348865, 0.9948237384571683, None)
```

Based on the above results, it appears that the multilabel classifier was *extremely* accurate, with precision scores, recall scores, and F-scores all above 0.99. While I am not 100% sure how to interpret the multilabel confusion matrix, here is my current understanding:

The first matrix above shows the output with respect to 0 (i.e. a given color is not in the color identity). There were no false positives, and only 6 false negatives. This matrix is heavily weighted in favor of true negatives, meaning predictions of "not-0" for an element of a multiple label which was "not-0".

The second matrix shows the output with respect to 1 (i.e. a given color is in the color identity). Again, the number of correct predictions (true positives and true negatives) is overwhelming. The most common type of error in this matrix is a false negative, which in this matrix means that there were about 100-130 instances where the correct label was 1 and the prediction was "not-1".

For this part, I only used unigrams, and the only method of preprocessing employed in the pipeline was the TfidfVectorizer. This vectorizer first converts each corpus (in this case, each of the "text" and "flavor_text" columns) into a document-term matrix, then removes stopwords according to scikit-learn's inbuilt set of stopwords. It then calculates the TF-IDF (term frequency-inverse document frequency) value for each remaining word. These scores are then used as features to predict and classify instances in this data set (in this case, to assign color labels to different Magic: The Gathering cards). The two confusion matrices and the associated performance metrics I achieved indicate that even this minimal amount of preprocessing was sufficient to train highly accurate classifier. The accuracy may also be due in part to the fraction of the data used for training and testing. I used a 75/25 training-test split for both classifiers in this section; it is possible that a different ratio could have produced lower levels of precision or recall.

---

## 3 Part III: Regression

To accomplish this task, I used the GridSearchCV function available through scikit-learn. I selected four regressor models (LASSO regression, Decision Tree, K-Nearest Neighbors, and Random Forest), specified certain values to test for different parameters, and then fit the models to the data to find the most accurate regressor. My target variable is `edhrec_rank`, which I understand encodes the popularity of a Magic: The Gathering card at the time the dataset was compiled. My feature variables were as follows: "text", "flavor_text", dummies encoding the presence of a specific color label in the "color identity" column, and dummies encoding either "uncommon" or "rare" values of the "rarity" column. I preprocessed the text columns with the TfidfVectorizer used in part II; the other feature columns exclusively take on values of 0 or 1, and so do not require preprocessing.

[31]:
```
estimator = joblib.load('regression_GridSearch.pkl')
```

[32]:
```
mtg = mtg.dropna(subset = ['flavor_text', 'text', 'color_identity',␣
 ↪'edhrec_rank', 'rarity']).reset_index(drop=True)
mtg = pd.get_dummies(mtg, columns = ['rarity'])
```

[33]:
```
#Specify X and Y
y = mtg['edhrec_rank']

X = mtg[['text','flavor_text','is_B','is_G', 'is_R', 'is_U', 'is_W',␣
 ↪'rarity_uncommon', 'rarity_rare']]
```

[34]:
```
estimator.fit(X,y)
```

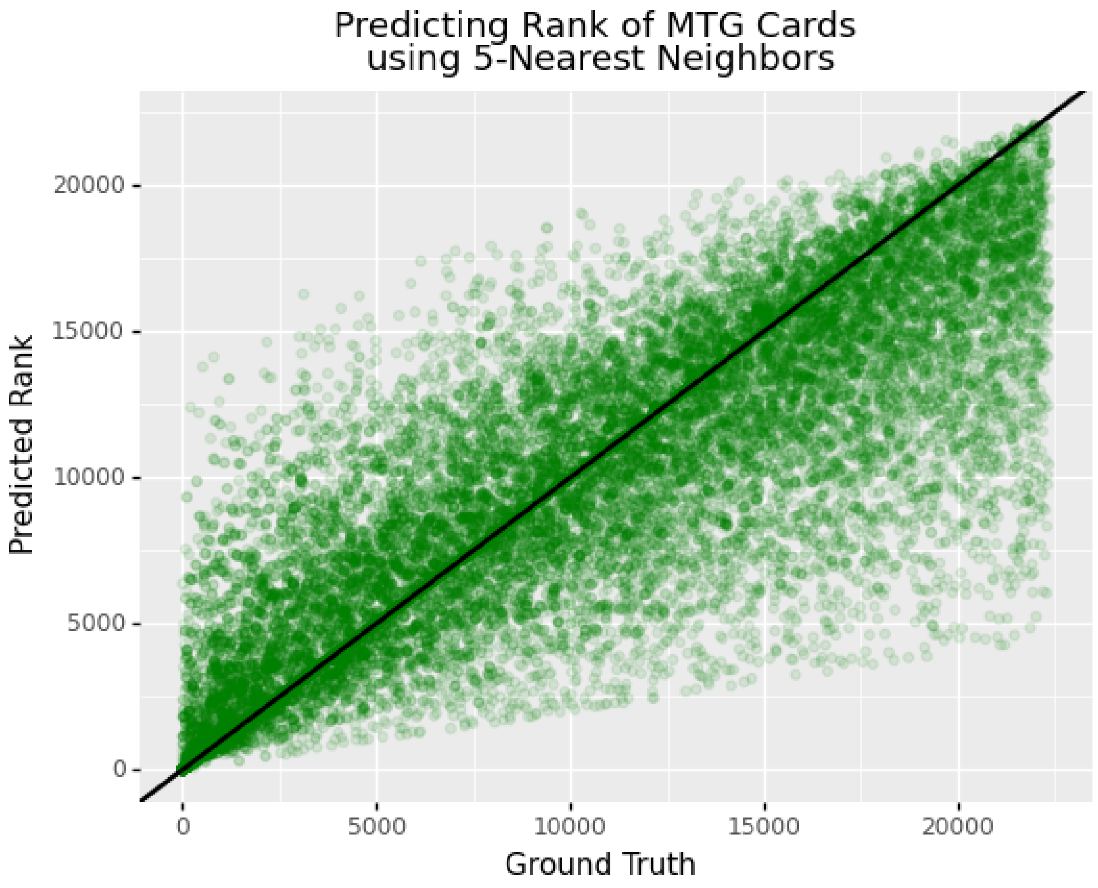
[34]:
```
Pipeline(steps=[('pre_process',
                 ColumnTransformer(transformers=[('text', TfidfVectorizer(),
                                                  'text'),
                                                 ('flavor_text',
                                                  TfidfVectorizer(),
                                                  'flavor_text')])),
                ('model', KNeighborsRegressor())])
```

[35]:
```
regressor_predictions = estimator.predict(X)
```

[36]:
```
plotting_data = pd.DataFrame({'actual': y,
                              'predicted': regressor_predictions})
```

[37]:
```python
from plotnine import *
```

[38]:
```python
(
    ggplot(plotting_data, aes(x='actual', y='predicted'))
    + geom_point(alpha = 0.1,
                 color = 'green')
    + geom_abline(intercept=0, # set the y-intercept value
                  slope=1,     # set the slope value
                  color = 'black',
                  size = 1
                 )
    + labs(x='Ground Truth',
           y='Predicted Rank',
           title='Predicting Rank of MTG Cards \nusing 5-Nearest Neighbors')
)
```



[38]:
```
<ggplot: (156700137003)>
```

The above graph plots the predicted ranks against the actual values provided in the dataset. The black diagonal line indicates perfect performance, where true value = prediction. Clearly, there is substantial variation, though it seems to follow a normal distribution both above and below the black line. In the process of testing the pipeline for this part, I found that K-Nearest Neighbors where k = 5, the strongest performing regressor in the Grid Search, produced a negative mean squared error of -2280000, at least. Even the most accurate methods I tested were largely inaccurate. The two possible conclusions I could draw from this are: -1: The features I selected have limited predictive power and do not contribute strongly to or majorly affect the rank of a given card, and -2: The specific regressor models I chose are not well suited to this data set.

In either case, the solution is effectively further experimentation, though such experimentation is not without possible challenges. Regarding the feature variables, I selected these ones because they had relatively low levels of missingness. Other variables like "power" may be more predictive, but "power" was also missing for over 14,000 of 29,000 observations.

As for the models themselves, I am surprised that the K-Nearest Neighbors regressor performed the best, and that it achieved optimal performance with only 5 neighbors. I expected the Random Forest regressor to perform better; that has been the case with other data tasks and projects I have worked on. If a LASSO regressor did not perform well, then a ridge regressor likely will not perform well either. Other methods like boosting and bootstrapping could increase the viability of some of the tree based methods, but this is uncertain.

## 4 Appendix A: BERTopic Code

```
[ ]: #Filename: topic_model.py
     from bertopic import BERTopic
     import pandas as pd

     #Read in data, drop NAs, reset index
     mtg = (pd.read_feather('../../../data/mtg.feather')).dropna(subset =␣
      ↪['flavor_text']).reset_index(drop=True)
     #Instantiate BERT model
     topic_model = BERTopic()
     #Fit model to flavor text
     topics, probs = topic_model.fit_transform(mtg['flavor_text'])
     #Save model
     topic_model.save('hw2_bert_model')
```

## 5 Appendix B: Multiclass Code

```
[ ]: import joblib
     import numpy as np
     import pandas as pd
     from sklearn.compose import ColumnTransformer
     from sklearn.feature_extraction.text import TfidfVectorizer
     from sklearn.model_selection import train_test_split
     from sklearn.pipeline import Pipeline
     from sklearn.svm import LinearSVC

     #Read in data
     mtg = (pd.read_feather('../../../data/mtg.feather'))
     #Drop rows with missing values in the variables of interest
     mtg = mtg.dropna(subset = ['flavor_text', 'text', 'color_identity']).
      ↪reset_index(drop=True)

     #Create numeric labels based on values of colors
     #New values: X = multiclass, Z = NaN
     mtg['color_label'] = [np.array(['X']) if len(x) > 1 else x for x in␣
      ↪mtg['color_identity']]
     mtg['color_label'] = [np.array(['Z']) if len(x) == 0 else x for x in␣
      ↪mtg['color_label']]
     mtg['color_label'] = np.concatenate(mtg['color_label'])

     #Merge labels into MTG data frame
     labels = pd.DataFrame(mtg['color_label'].unique()).reset_index()
     #Add one because zero indexed
     labels['index'] = labels['index']+1
     labels.columns = ['label', 'color_label']
     mtg = labels.merge(pd.DataFrame(mtg), how = 'right', on = 'color_label')

     #Select labels as targets
     y = mtg['label']

     #Select text columns as features
     X = mtg[['text', 'flavor_text']]

     #Training test split 75/25
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)

     #Preprocess text (this took several hours to debug and I am honestly not joking)
     preprocess = ColumnTransformer(transformers=[('text', TfidfVectorizer(), 'text'),
                                                 ('flavor_text', TfidfVectorizer(),␣
      ↪'flavor_text')])

     #Create pipeline with preprocessing and linear SVC
```

```
pipe = Pipeline([
    ('preprocess', preprocess),
    ('LinearSVC', LinearSVC())
])

#Fit pipe to training data
fitted_pipe = pipe.fit(X_train, y_train)

#Export pickeled pipe
joblib.dump(fitted_pipe, 'multiclass_pipe.pkl')
```

## 6 Appendix C: Multilabel Code

```
[ ]: import joblib
     import numpy as np
     import pandas as pd
     from sklearn.compose import ColumnTransformer
     from sklearn.feature_extraction.text import TfidfVectorizer
     from sklearn.model_selection import train_test_split
     from sklearn.preprocessing import MultiLabelBinarizer
     from sklearn.pipeline import Pipeline
     from sklearn.svm import SVC
     from sklearn.multiclass import OneVsRestClassifier

     #Read in data
     mtg = (pd.read_feather('../../../data/mtg.feather'))
     #Drop rows with missing values in the variables of interest
     mtg = mtg.dropna(subset = ['flavor_text', 'text', 'color_identity']).
      ↪reset_index(drop=True)

     for letter in np.unique(np.concatenate(np.array(mtg['color_identity']))):
         mtg["is_"+letter] = [1 if letter in x else 0 for x in mtg['color_identity']]

     letters = mtg.columns.str.contains('is_')

     mtg['labels'] = mtg[mtg.columns[letters]].values.tolist()

     #Select labels as targets
     mlb = MultiLabelBinarizer()
     y = mlb.fit_transform(mtg['labels'])
     #Select text columns as features
     X = mtg[['text', 'flavor_text']]

     #Training test split 75/25
     X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.25)

     #Preprocess text (this took several hours to debug and I am honestly not joking)
     preprocess = ColumnTransformer(transformers=[('text', TfidfVectorizer(), 'text'),
                                                  ('flavor_text', TfidfVectorizer(),␣
      ↪'flavor_text')])

     #Create pipeline with preprocessing and linear SVC
     pipe = Pipeline([
         ('preprocess', preprocess),
         ('classifier', OneVsRestClassifier(SVC()))
     ])

     #Fit pipe to training data
```

```python
fitted_pipe = pipe.fit(X_train, y_train)

#Export pickeled pipe
joblib.dump(fitted_pipe, 'multilabel_pipe.pkl')
```

## 7 Appendix D: Rank Regressor

[ ]:
```
import joblib
import numpy as np
import pandas as pd
from sklearn.compose import ColumnTransformer
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.model_selection import train_test_split, GridSearchCV, KFold
from sklearn.pipeline import Pipeline
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.linear_model import Lasso

#Read in data
mtg = (pd.read_feather('../../../data/mtg.feather'))
#Drop rows with missing values in the variables of interest
mtg = mtg.dropna(subset = ['flavor_text', 'text', 'color_identity',␣
 ↪'edhrec_rank', 'rarity']).reset_index(drop=True)

#Create dummy columns based on color labels
for letter in np.unique(np.concatenate(np.array(mtg['color_identity']))):
    mtg["is_"+letter] = [1 if letter in x else 0 for x in mtg['color_identity']]

#Feature variables: text, flavor text, color labels, rarity
mtg = pd.get_dummies(mtg, columns = ['rarity'])

#Specify X and Y
y = mtg['edhrec_rank']

X = mtg[['text','flavor_text','is_B','is_G', 'is_R', 'is_U', 'is_W',␣
 ↪'rarity_uncommon', 'rarity_rare']]

#70/30 training-test split
X_train, X_test, y_train, y_test = train_test_split(X,y, test_size = 0.3)

#(1) Choose a number of folds and specify a random state
fold_generator = KFold(n_splits=5, shuffle=True)

#(2) Specify a preprocessing step for the text columns
preprocess = ColumnTransformer([('text', TfidfVectorizer(), 'text'),
                                ('flavor_text', TfidfVectorizer(),␣
 ↪'flavor_text')])

#(3) Create the model pipe
pipe = Pipeline(steps=[('pre_process', preprocess),
                       ('model',None)])
```

```
#(4) Instantiate the search space
search_space = [
    # Naive Bayes Classifier
    {'model' : [Lasso()]},

    # K-Nearest-Neighbors, also specifying values of K to test
    {'model' : [KNeighborsRegressor()],
    'model__n_neighbors':[5,10,25,50]},

    # Decision Tree, also specifying depth levels to test
    {'model': [DecisionTreeRegressor()],
    'model__max_depth':[2,3,4]},

    # Random forest, also specifying depth levels, numbers of estimators, and␣
 ↪numbers of features to test
    {'model' : [RandomForestRegressor()],
    'model__max_depth':[2,3,4],
    'model__n_estimators':[500,1000,1500],
    'model__max_features':[3,4,5]},
]

#Assemble the GridSearch
search = GridSearchCV(pipe,
                      search_space,
                      cv = fold_generator,
                      scoring='neg_mean_squared_error',
                      n_jobs=-1)

#Fit the GridSearch
search.fit(X_train, y_train)

#Save the best estimator
joblib.dump(search.best_estimator_, 'regression_GridSearch.pkl')
```