Table of Contents

In [1]:
```
import pandas as pd
import numpy as np
# rng = np.random.default_rng(2)

# import holoviews as hv
# from holoviews import opts
# hv.extension('bokeh')
```

In [2]:
```
!dvc pull
```

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

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
  1. the `topic_visualization` interactive plot see docs
  2. 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.
  3. Once you have names, create a *Dynamic Topic Model* by following their documentation. Use the `release_date` column as timestamps.
  4. Describe what you see, and any possible issues with the topic models BERTopic has created. **This is the hardest part... interpreting!**

In [3]:
```
# Read in magic data
df = (
    pd.read_feather('../../../data/mtg.feather')
    .dropna(subset=['flavor_text', 'text'])
)
```

In [4]:
```
# Load trained BERTopic model
from bertopic import BERTopic

# topic_model = BERTopic.load("bertopic_model")
# topic_model = BERTopic.load("my_model_custom_embeddings")
# topic_model = BERTopic.load("my_model_no_min_topic_size")
topic_model = BERTopic.load("my_model_embeddings")
```

I did a number of different iterations when training a BERTopic model. The last one is the one I decided to use for this submission.

I did the following:

- Preprocess the flavor text to decontract words ("won't" changed to "will not", for example)
- Created a custom embedding model using SentenceTransfomer that I trained on the flavor_text corpus itself (See "Custom Embeddings")
- Included a CountVectorizer inside BERTopic to include English stopwords and ngram_range = (1,1) (See "I am I facing memory issues. Help!")
- Set min_cluster_size in HDBSCAN equal to 100 (See "How do I reduce topic outliers")
- Set nr_topic to 'auto' so BERTopic can merge topics that are similar to one another

Topic Visualization

In [5]:
```
from my_functions import preprocess

# Preprocess the flavor_text
docs = preprocess(df.flavor_text)

# Fit transform
topics, probs = topic_model.fit_transform(docs)

topic_model.visualize_topics()
```

```
100%|██████████████████████████████████| 29635/29635 [00:00<00:00, 100179.16
it/s]
Batches:   0%|          | 0/927 [00:00<?, ?it/s]
OMP: Info #273: omp_set_nested routine deprecated, please use omp_set_max_a
ctive_levels instead.
```

```
huggingface/tokenizers: The current process just got forked, after parallel
ism has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(t
rue | false)
huggingface/tokenizers: The current process just got forked, after parallel
ism has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(t
rue | false)
huggingface/tokenizers: The current process just got forked, after parallel
ism has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(t
rue | false)
huggingface/tokenizers: The current process just got forked, after parallel
ism has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(t
rue | false)
huggingface/tokenizers: The current process just got forked, after parallel
ism has already been used. Disabling parallelism to avoid deadlocks...
To disable this warning, you can either:
        - Avoid using `tokenizers` before the fork if possible
        - Explicitly set the environment variable TOKENIZERS_PARALLELISM=(t
rue | false)
```

In [6]:
```
len(topic_model.get_topics())
```

Out[6]: 46

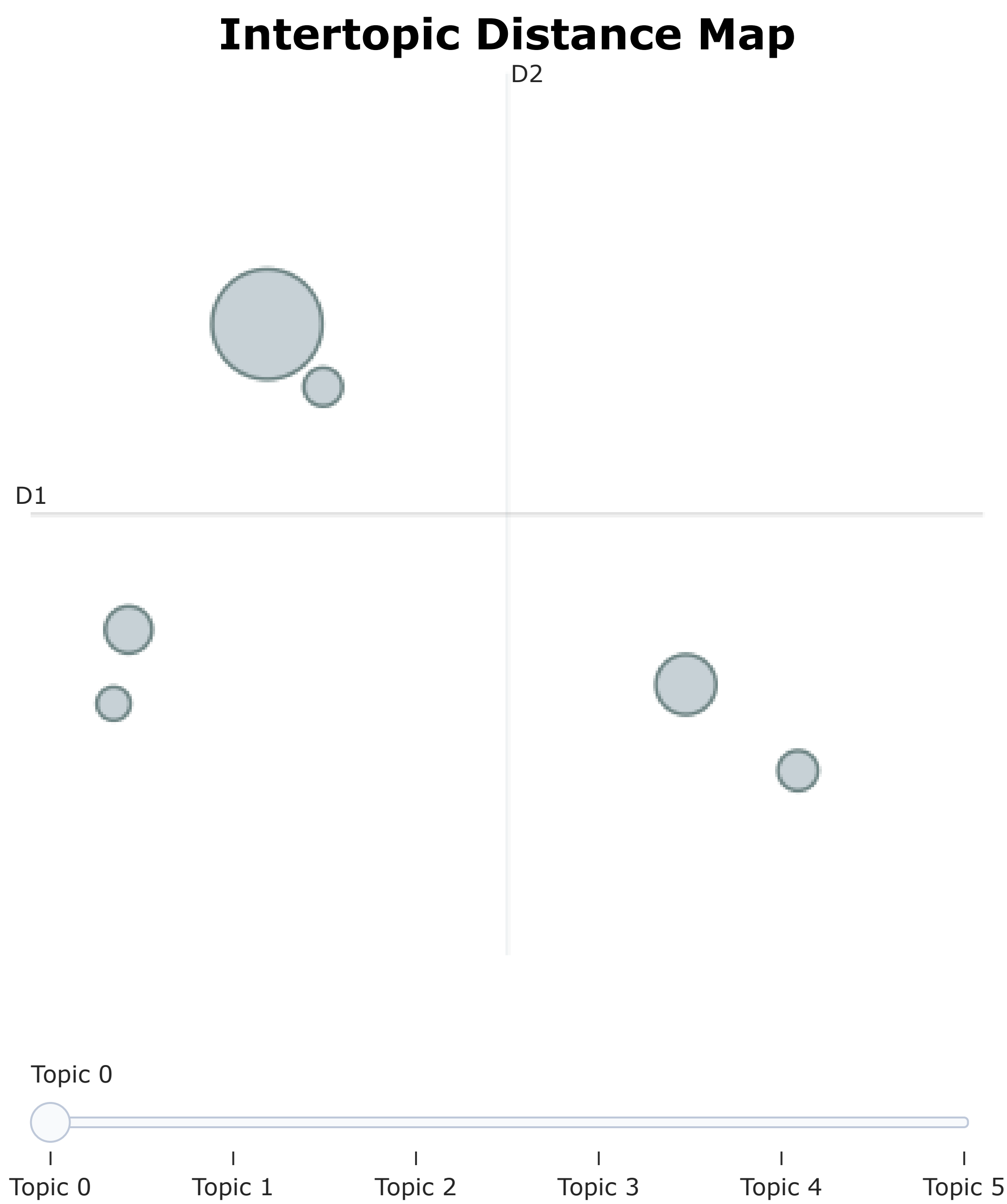
Because BERTopic is highly stochastic nature of UMAP (See "Why are the results not consistent between runs?") it took me a while to figure out the general layout of the topics.

Thanks to the intertopic distance chart, I could see 6 somewhat-distinct clusters in the topics. Therefore, I decided to further reduce the number of topics before I have to name them.

Topic Reduction after Training

See "Topic Reduction"

In [7]:
```
new_topics, new_probs = topic_model.reduce_topics(docs, topics, nr_topics=6)
topic_model.visualize_topics()
```



In [8]:
```
set(new_topics)
```

Out[8]:
```
{-1, 0, 1, 2, 3, 4, 5}
```

Topic Names

In [9]:
```
topic_model.get_topic_info()
```

Out [9]:

	Topic	Count	Name
0	-1	18869	-1_life_death_like_world
1	0	5893	0_power_dead_nature_strength
2	1	1752	1_fight_sword_blade_battle
3	2	1065	2_sun_light_darkness_night
4	3	782	3_prey_hunt_hunter_werewolves
5	4	724	4_mage_magic_wizard_mages
6	5	550	5_hear_silent_roar_sound

From the reduced topics above, these are the names I've come up with the following names:

Topic -1: outliers.

Topic 0: Earth/ Nature

Topic 1: Death

Topic 2: Elves/ Forest Creatures

Topic 3: Wolves/ Hunters

Topic 4: Godly Gifts

Topic 5: War

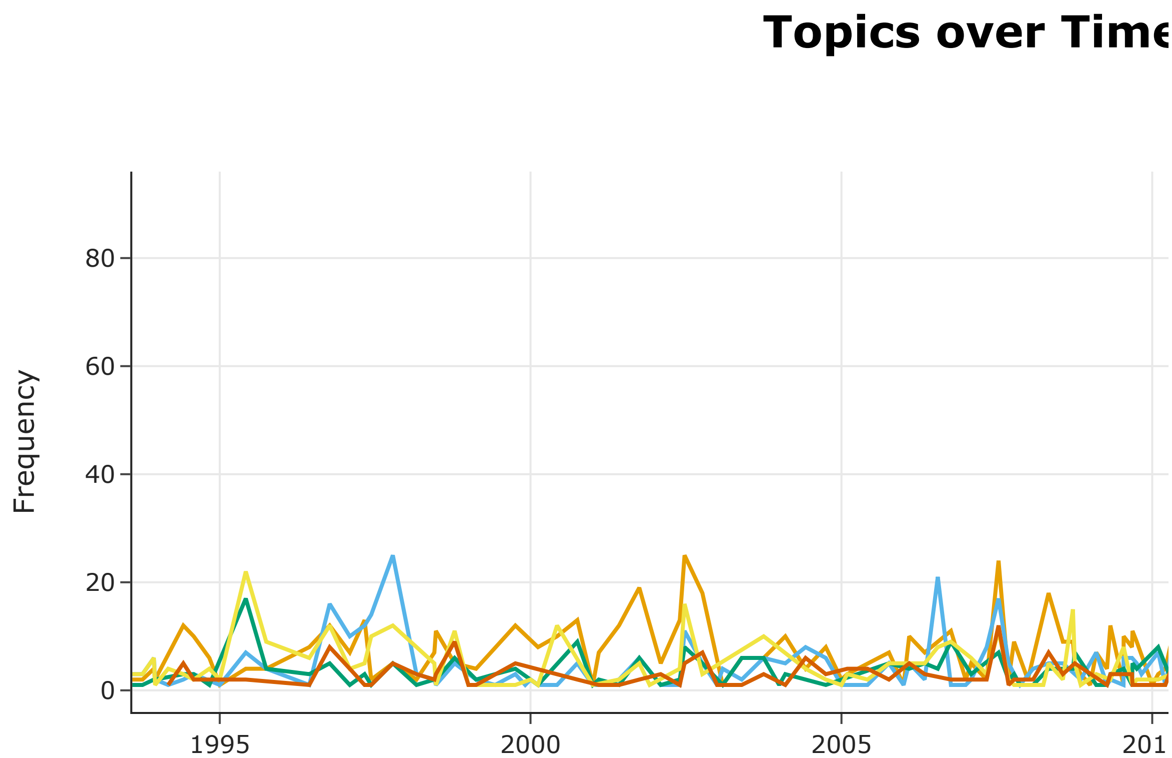
Dynamic Topic Models

In [10]:
```
# Convert release_date to a list
timestamps = df.release_date.tolist()
```

In [11]:
```
# Using new_topics because it's been reduced to 6 topics
topics_over_time = topic_model.topics_over_time(docs, new_topics, timestamps
```

```
281it [00:05, 49.72it/s]
```

In [12]:
```
topic_model.visualize_topics_over_time(topics_over_time, topics = [1,2,3,4,5
```



Comments

I spent a lot of time on part 1 because I wanted to play with the parameters and really understand BERTopic. Thanks to me implementing a custom embedding on the corpus and a second layer topic reduction based on the clusters I observed, I think the final product (the Dynamic Topic Models) chart doesn't seem to outrageous.

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?

Preprocessing

The preprocessing steps I did in both multiclass.py and multilable.py are the same as the steps I did for flavor_text in topic_model.py. The preprocess function for the text column(s) came from my_functions.py to ensure that I used the same preprocessing steps in all parts of this notebook, which includes:

- Decontracting words - for example, turning "won't" to "will not" and "I'll" to "I will"
- I did not turn words into lowercase because I did not want to lose names of people and places from the data
- I chose to decontract words instead of just stripping special character because I thought "ill" (as in illness) in this Magic dataset shouldn't get clumped together with "I will" ("I'll" lowercased and stripped)
- I also stripped special escapes like \r and \n from the text

After apply the preprocess() functions on the text and flavor_text columns, I concatenated both columns (will be shown below) - this is my X features.

In the multiclass case, because I needed to consider cards with more than one color identities an unlabeled card, I needed to drop them (along with cards with no color identity [ ]) so I could fit a multiclass classifier on the color_identity column.

In the multilabel case, I applied a MultiLabelBinarizer on the color_identity column. This way, I did not need to drop cards that had no color_identity - they would simply turn to an array of [0, 0, 0, 0, 0] (since we have 5 colors), whereas a cards with 4 different colors might look something like [1, 0, 1, 1, 1]

Multiclass

In [13]:
```
# Read in magic data
df = (
    pd.read_feather('../../../data/mtg.feather')
    .dropna(subset=['flavor_text', 'text'])
    .reset_index(drop=True)
)
```

In [14]:
```
# Keep rows where the len of color_identity is 1
df = df[df['color_identity'].map(lambda d: len(d)) == 1].reset_index(drop=Tr

# Because all lists in df.color_identity now has length 1, take the first it
y = df.color_identity.apply(lambda x: x[0])

y.shape
```

Out[14]:
```
(22418,)
```

In [15]:
```
from my_functions import preprocess

# Preprocess text columns
clean_flavor_text = preprocess(df.flavor_text)
clean_text = preprocess(df.text)

X = []

# Concatenate the 2 text columns together
for i in range(len(clean_text)):
    text_concat = clean_text[i] + ". " + clean_flavor_text[i]
    X.append(text_concat)

len(X)
```

```
100%|██████████████████████████████████| 22418/22418 [00:00<00:00, 101603.13
it/s]
100%|███████████████████████████████████| 22418/22418 [00:00<00:00, 93638.18
it/s]
```

Out[15]:
```
22418
```

In [16]:
```
# load trained model
import pickle

with open('multiclass.pkl', 'rb') as f:
    multiclass = pickle.load(f)
```

In [17]:
```
from sklearn.model_selection import train_test_split

# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, ra

#fit model with training data
multiclass.fit(X_train, y_train)

#evaluation on test data
y_pred = multiclass.predict(X_test)
```

Confusion matrix

In [18]:
```
from sklearn.metrics import confusion_matrix

confusion_matrix(y_pred, y_test)
```

Out[18]:
```
array([[ 991,   29,   29,   27,   34],
       [  24, 1046,   36,   13,   31],
       [  29,   24, 1000,   13,   34],
       [  18,   39,   17, 1009,   30],
       [  39,   23,   30,   32, 1008]])
```

Classification report

In [19]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           B       0.89      0.90      0.90      1101
           G       0.91      0.90      0.91      1161
           R       0.91      0.90      0.90      1112
           U       0.91      0.92      0.91      1094
           W       0.89      0.89      0.89      1137

    accuracy                           0.90      5605
   macro avg       0.90      0.90      0.90      5605
weighted avg       0.90      0.90      0.90      5605
```

Multilabel

In [20]:
```
# Read in magic data
df = (
    pd.read_feather('../../../data/mtg.feather')
    .dropna(subset=['flavor_text', 'text'])
    .reset_index(drop=True)
)
```

In [21]:
```
from sklearn.preprocessing import MultiLabelBinarizer

MLB = MultiLabelBinarizer()
y = MLB.fit_transform(df.color_identity)

MLB.classes_
```

Out[21]:
```
array(['B', 'G', 'R', 'U', 'W'], dtype=object)
```

In [22]:
```
from my_functions import preprocess

clean_flavor_text = preprocess(df.flavor_text)
clean_text = preprocess(df.text)

X = []
for i in range(len(clean_text)):
    text_concat = clean_text[i] + ". " + clean_flavor_text[i]
    X.append(text_concat)

len(X)
```

```
100%|████████████████████████████████| 29635/29635 [00:00<00:00, 101270.67
it/s]
100%|█████████████████████████████████| 29635/29635 [00:00<00:00, 90429.14
it/s]
```

Out[22]:
```
29635
```

In [23]:
```
# load trained model
import pickle

with open('multilabel.pkl', 'rb') as f:
    multilabel = pickle.load(f)
```

In [24]:
```
from sklearn.model_selection import train_test_split

# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, ra

#fit model with training data
multilabel.fit(X_train, y_train)

#evaluation on test data
y_pred = multilabel.predict(X_test)
```

Confusion matrix

In [25]:
```
from sklearn.metrics import multilabel_confusion_matrix

print(multilabel_confusion_matrix(y_test,y_pred))
```

```
[[[5723  151]
  [ 197 1338]]

 [[5615  159]
  [ 201 1434]]

 [[5668  140]
  [ 188 1413]]

 [[5715  178]
  [ 167 1349]]

 [[5544  204]
  [ 179 1482]]]
```

Classification report

In [26]:
```
from sklearn.metrics import classification_report

print(classification_report(y_test, y_pred,target_names=MLB.classes_,zero_di
```

```
              precision    recall  f1-score   support

           B       0.90      0.87      0.88      1535
           G       0.90      0.88      0.89      1635
           R       0.91      0.88      0.90      1601
           U       0.88      0.89      0.89      1516
           W       0.88      0.89      0.89      1661

   micro avg       0.89      0.88      0.89      7948
   macro avg       0.89      0.88      0.89      7948
weighted avg       0.89      0.88      0.89      7948
 samples avg       0.92      0.90      0.87      7948
```

Comments

I am actually blown away by how well both models did on the test set. Both managed to achive high precision, recall and f1-scores. I feel like with results *this* good, my spidey senses should be firing off. I really want to know what I did wrong in the preprocessing pipelines that returned these results.

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?

Comments:

I tried a lot of things, but the ones that I ended up going with are:

- Created sparse matrices for types, subtypes, super_types, color_identity using multilabelbinarizer (which ended up working)
- One-Hot encoded block and rarity
- Applied MinMaxScalre to converted_mana_cost
- GridSearch a bunch of different regressors

Things that didn't work:

- I really tried to scale the output feature, first using np.log, then with TransformedTargetRegressor, but I couldn't get the pipeline to work right
- I tried adding the TFIDF for the text & flavor_text columns, but they made the matrix really big and my computer couldn't handle it (and I have a computer with 10 CPU cores and 32GB or RAM, so if mine can't do it, I doubt most of my classmate's laptops can)

If I had more time, I would've liked to:

- Done some form of dimensionality reduction with PCA
- Throw a neural net on top of this with keras. I've done a little bit NLP with neural nets before, so I was a little stumped when I realized that sklearn couldn't easily handle large matrices/ tensors - that took a while to get around.

In [27]:
```
# load trained model
import pickle

with open('regression.pkl', 'rb') as f:
    reg = pickle.load(f)
```

In [28]:
```
import pandas as pd
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.model_selection import train_test_split
from my_functions import multi

df = (
    pd.read_feather('../../../data/mtg.feather')
    .dropna(subset = ['edhrec_rank'])
    .reset_index(drop=True)
)
# Source: https://scikit-learn.org/stable/auto_examples/compose/plot_column_


numeric_features = ["converted_mana_cost"]
numeric_transformer = Pipeline(
    steps=[("scaler", MinMaxScaler())]
)

cat_features = ["block","rarity"]
cat_transformer = OneHotEncoder(handle_unknown="ignore")
multi_label = multi(df,["types","subtypes", "color_identity","supertypes"])

X = pd.concat([df[['converted_mana_cost','rarity',"block"]],multi_label], ax
y = df['edhrec_rank']

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25, ra
```

In [29]:
```
#fit model with training data
reg.fit(X_train, y_train)

#evaluation on test data
y_pred = reg.predict(X_test)

from sklearn.metrics import mean_squared_error, mean_absolute_error, r2_scor
print(mean_squared_error(y_test, y_pred),mean_absolute_error(y_test, y_pred)
```

```
22205662.935580887 3333.3070807808263 0.46898020211749236
```

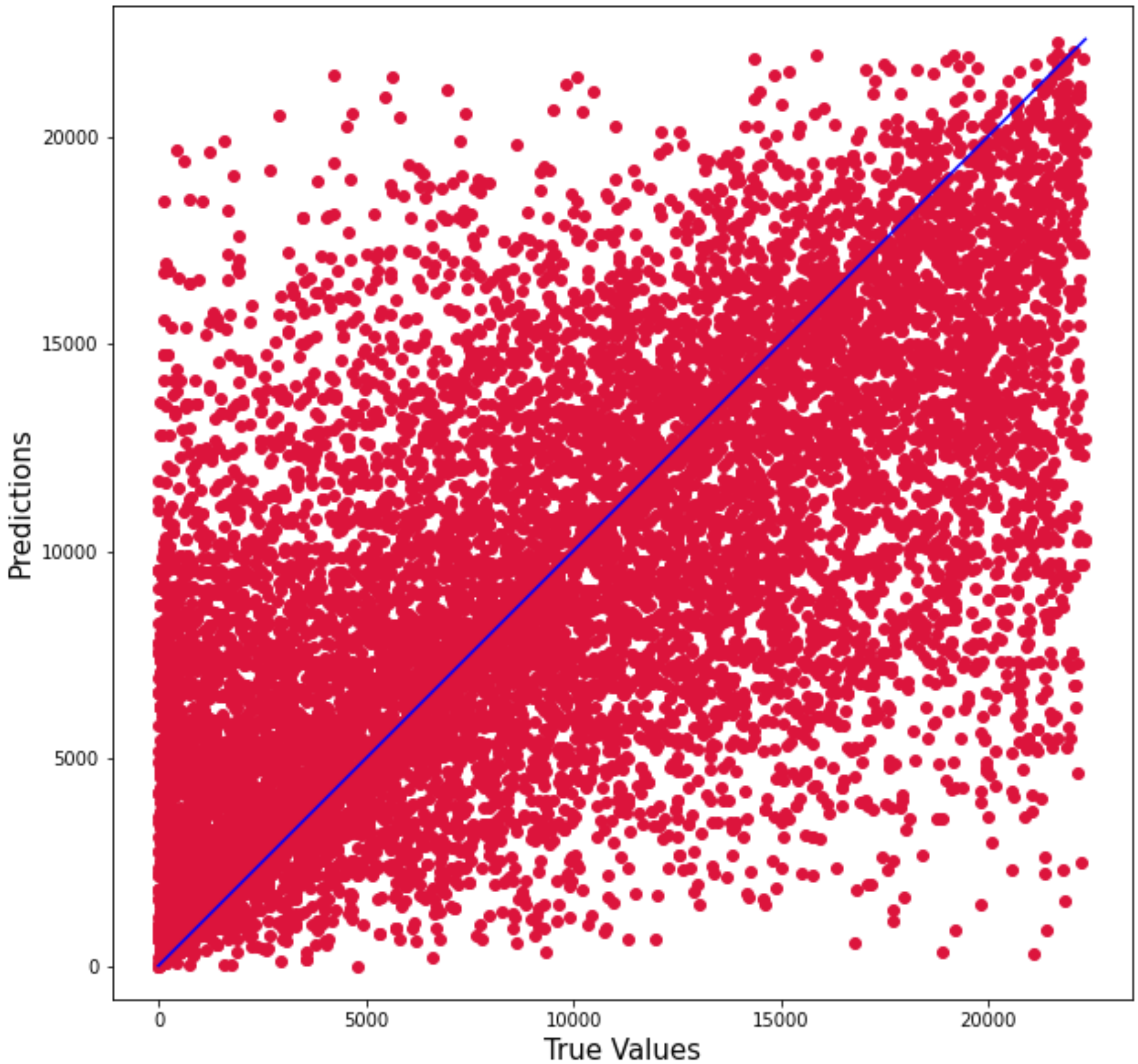
Actual vs Predicted Plot

In [30]:
```
import matplotlib.pyplot as plt
#Source: https://stackoverflow.com/questions/58410187/how-to-plot-predicted-



plt.figure(figsize=(10,10))
plt.scatter(y_test, y_pred, c='crimson')
# plt.yscale('log')
# plt.xscale('log')
p1 = max(max(y_pred), max(y_test))
p2 = min(min(y_pred), min(y_test))

plt.plot([p1, p2], [p1, p2], 'b-')
plt.xlabel('True Values', fontsize=15)
plt.ylabel('Predictions', fontsize=15)
plt.axis('equal')
plt.show()
```



Feature Importance

In [31]:
```
from sklearn.inspection import permutation_importance
vi = permutation_importance(reg,X_train,y_train,n_repeats=5)
```

In [32]:
```
# Organize as a data frame
vi_dat = pd.DataFrame(dict(variable=X_train.columns,
                           vi = vi['importances_mean'],
                           std = vi['importances_std']))

# Generate intervals
vi_dat['low'] = vi_dat['vi'] - 2*vi_dat['std']
vi_dat['high'] = vi_dat['vi'] + 2*vi_dat['std']

# But in order from most to least important
vi_dat = vi_dat.sort_values(by="vi",ascending=False).reset_index(drop=True)


vi_dat
```

Out[32]:

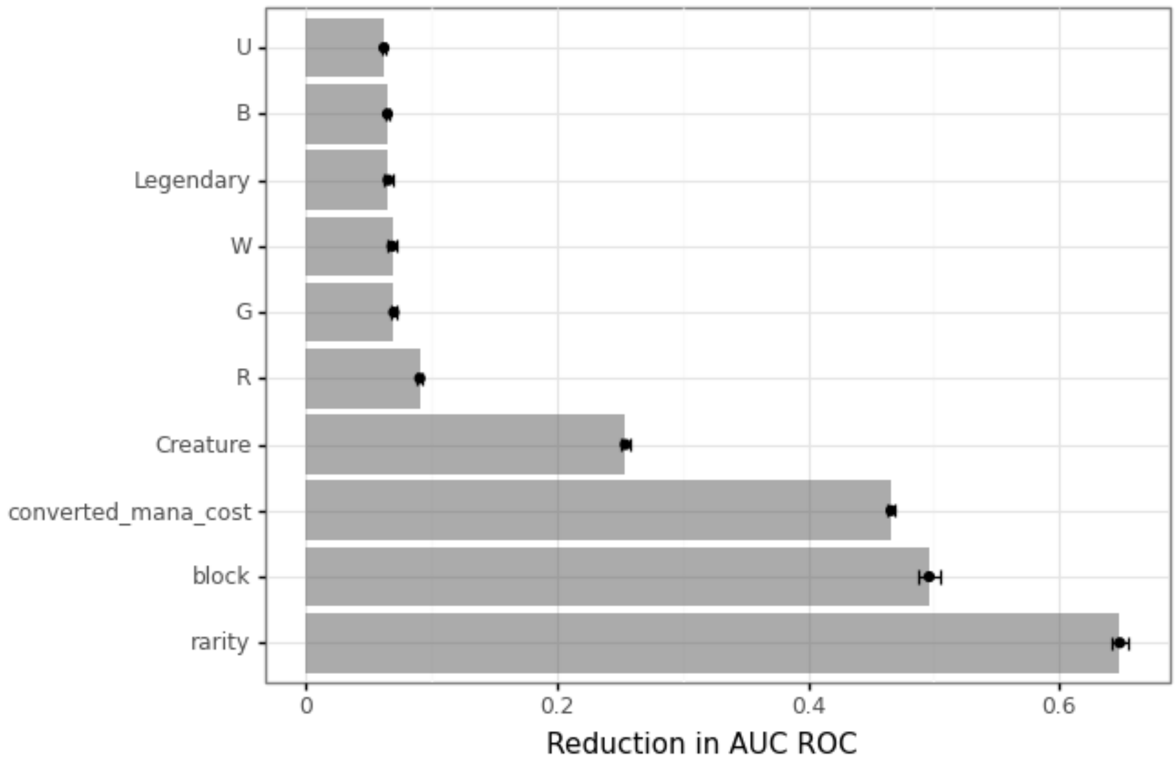
	variable	vi	std	low	high
0	rarity	6.482141e-01	0.003268	6.416790e-01	6.547493e-01
1	block	4.965136e-01	0.004339	4.878349e-01	5.051923e-01
2	converted_mana_cost	4.661245e-01	0.001588	4.629486e-01	4.693005e-01
3	Creature	2.546113e-01	0.001684	2.512425e-01	2.579802e-01
4	R	9.099565e-02	0.000885	8.922537e-02	9.276594e-02
...	...	...	...	...	...
349	Coward	4.645543e-08	0.000000	4.645543e-08	4.645543e-08
350	Koth	2.878551e-08	0.000000	2.878551e-08	2.878551e-08
351	Tyvar	2.308796e-08	0.000000	2.308796e-08	2.308796e-08
352	Fractal	0.000000e+00	0.000000	0.000000e+00	0.000000e+00
353	Nautilus	0.000000e+00	0.000000	0.000000e+00	0.000000e+00

354 rows × 5 columns

In [36]:
```
top10 = vi_dat[0:10]
```

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

# Plot
(
    ggplot(top10,
          aes(x="variable",y="vi")) +
    geom_col(alpha=.5) +
    geom_point() +
    geom_errorbar(aes(ymin="low",ymax="high"),width=.2) +
    scale_x_discrete(limits=top10.variable.tolist()) +
    coord_flip() +
    labs(y="Reduction in AUC ROC",x="")
)
```



Out[37]:
```
<ggplot: (827719078)>
```

Part 4:

I picked my multilabel model, which has already performed pretty well early-on , according to its F-1 score, precision and recall. In this experiment, I picked the label_ranking_score as the metrics to measure.

I wanted to tune the loss measurement and penalty function of `LinearSVC` 's. The default loss function is `squared_hinge` loss. In my experiment, I changed the default loss function to `hinge` and recorded the change in `metrics.json` . Initially I wanted to change the L2 norm to L1 norm as well, but L1 norm does not work with hinge loss.

In [14]:
```
!dvc exp diff
```

```
Path          Metric              HEAD     workspace    Change
metrics.json  label_ranking_loss  0.11119  0.11358      0.0023845

Path         Param           HEAD           workspace    Change
params.yaml  LinearSVC.loss  squared_hinge  hinge        diff not supported
```

Keeping the default L2 penalty and change the loss function to `hinge` adds 0.0023645 to the label ranking loss. The closer label ranking loss is to 0 the better so this is not optimal.

In [15]:
```
!dvc exp diff
```

```
Path          Metric              HEAD     workspace    Change
metrics.json  label_ranking_loss  0.11119  0.11244      0.0012485
metrics.json  use_idf             True     False        -1

Path         Param                         HEAD    workspace    Change
params.yaml  TfidfTransformer.use_idf      True    False        -1
```

Changing use_idf to False in TFIDF added more to label_ranking_loss

In [16]:
```
!dvc exp diff
```

```
Path          Metric              HEAD     workspace    Change
metrics.json  label_ranking_loss  0.11119  0.13832      0.027129

Path         Param                                 HEAD    workspace    Change
params.yaml  CountVectorizer.ngram_range.min_n     1       2            1
```

And changing the range of the ngram in countvectorizer to (2,2) instead of (1,2) also made the multilabel classifier worse

In [19]:
```
!dvc exp diff
```

```
Path          Metric              HEAD     workspace    Change
metrics.json  label_ranking_loss  0.11119  0.10573      -0.0054663

Path         Param                                 HEAD    workspace    Change
params.yaml  CountVectorizer.ngram_range.max_n     2       3            1
```

However, the one thing that did improve the classifier is changing the ngram of countvectorizer to (1,3) instead of (1,2)

In [ ]: