Report:
Predicting Kickstarter Campaign Outcomes Using NLP

Introduction:

For my second capstone, my goal was to predict campaign success on Kickstarter by leveraging NLP techniques as well as recently developed practices in machine learning in order to organize data processing and modeling code.

Data was collected from a site hosted by Webrobots, a group that scrapes Kickstarter data monthly and hosts each month as a series of csv files. In order to analyze and create training-testing data the csv files needed to be collected and loaded into a single database and then queried.

Once the training and testing data was created I then applied various NLP techniques to create text features which, after being combined with additional features (numeric and categorical) like campaign target and creation date, were then fed through various models to understand how much signal was present.

Ultimately while showing the viability of NLP feature engineering in predicting the outcome of Kickstarter campaigns within the first two weeks of a campaign, additional data and campaign attributes would most likely have lifted the average model accuracy above 65%.

Key Stakeholders:

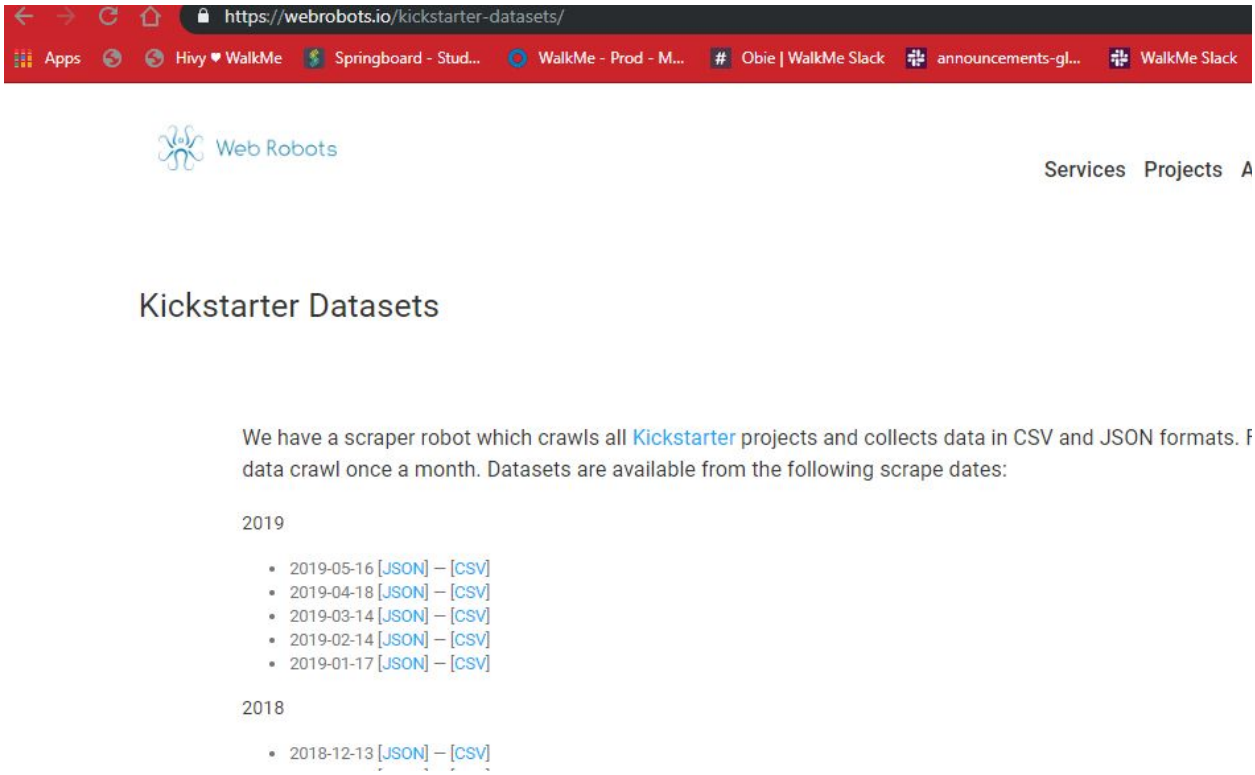
Potential parties that could be interested in this project include:

1. Potential creators wanting to understand:
    - 1) What is their realistic chance of being successful on Kickstarter
    - 2) What does the competitive landscape look like for similar categories, etc
    - 3) How they can better message their products.
2. Knock-off manufacturers wanting to understand who they should be ripping off
3. Competitive crowdfunding sites wanting to understand:
    - 1) The supply & demand of crowdfunding sites
    - 2) The products being moved (or not)

Data Overview:

Data for the project came from a series of csv files (hosted by Webrobots) which were produced by scraping the Kickstarter site once a month (https://webrobots.io/kickstarter-datasets/).

Each scrape could produce anywhere from 25-50+ csv files which were stored in folders labeled by the month and day the data was scraped. Each row in a csv file represents a single project and includes features like the creator, goal amount, country of origin and important text fields like “blurb” (a 1 sentence summary of the project), “category”, “title” and “creator”. Given the scrapes were performed once a month and the average campaign length was 30 days, the same campaign could be captured multiple times and a majority of campaigns could have been posted between scrapes.



A concern of using data from a point in time too far into the campaign is producing a model that results in overly high accuracy because of signal words like “success”, “fail” or even phrases like “reached our goal ahead of schedule!”.

Given we’re trying to predict a specific outcome (“successful” or “failed”) and we have labeled data, this would be a classification problem that could be addressed using a logistic regression, random forest classifier, or similar algorithms. Our goal is to use as many of the predictors as possible, given each file contains only about 32 features. We have a mix of data types including datetime columns (deadline, state_changed_at, created_at), numeric columns (backers, goal, usd_pledged), text (blurb, title), and categorical (country, currency, state).

Deliverables will include a jupyter notebook, a summary paper, and a slide deck. The comprehensive list of possible features were:

- 'backers_count', 'blurb', 'category',
  'converted_pledged_amount', 'country', 'created_at', 'creator',
  'currency', 'currency_symbol', 'currency_trailing_code',
  'current_currency', 'deadline', 'dirname', 'disable_communication',
  'friends', 'fx_rate', 'goal', 'id', 'is_backing', 'is_starrable',

'is_starred', 'last_update_published_at', 'launched_at', 'location',
'name', 'permissions', 'photo', 'pledged', 'profile', 'slug',
'source_url', 'spotlight', 'staff_pick', 'state', 'state_changed_at',
'static_usd_rate', 'unread_messages_count', 'unseen_activity_count',
'urls', 'usd_pledged', 'usd_type'

In total the original data set was about 30.6 MB and consisted of projects scraped between Jan 28, 2016 and Feb 14, 2019. Some important considerations needed to be handled given the data collection method.

1) Projects would be duplicated between scrapes and we wanted the earliest instance as opposed to the duplicates.
2) I was only interested in using data from projects within the early days of the campaign in order to avoid data leakage.
3) Many of the columns had data that was nested in a JSON format and needed to be split out.
    a) Ex:

{"id":45,"name":"Art Books","slug":"publishing/art books","position":3,"parent_id":18,"color":14867664,"urls":{"web":{"discover":"http://www.kickstarter.com/discover/categories/publishing/art%20books"}}}

Because of the timing of the data collection, we won't be able to answer questions like:

- How did the campaigns perform over time?
- Are there differences in success factors between categories?
- Are there regional differences?
- How were edits handled within the same campaigns over the lifetime of the campaign?

Data Acquisition & Processing:

In order to begin cleaning the data I needed to first compile the 30.6MB of data spread across 100+ csv files into a single source that could be queried. I decided to leverage the sqlite3 & os python packages to create a local SQLite database that could be loaded with data from the csv files, which would allow me to define the schema, query the records, and align the common columns.

After defining the table schema, I also converted the datetime columns (state_changed_at_clean, created_at_clean, deadline_clean, launched_at_clean) from a utc timestamp to a human readable format. Once all the projects were loaded into a single table, I then needed to stage the creation of the de-duped master data set.

Specifically, I selected projects that were posted within less than 15 days prior to a scrape being run. This list would consist of the project outcomes and would have the duplicates dropped from the pandas dataframe produced from running the query.

```
In [20]: 1 project_outcomes.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 1885073 entries, 0 to 1885072
Data columns (total 10 columns):
Unnamed: 0                int64
id                        int64
slug                      object
creator                   object
created_at_clean          object
launched_at_clean         object
profile                   object
country                   object
state                     object
state_changed_at_clean    object
dtypes: int64(2), object(8)
memory usage: 143.8+ MB
```

```
In [19]: 1 project_starting.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 478 entries, 0 to 477
Data columns (total 12 columns):
Unnamed: 0           478 non-null int64
id                   478 non-null int64
slug                 478 non-null object
name                 478 non-null object
blurb                478 non-null object
category             478 non-null object
goal                 478 non-null int64
launched_at_clean    478 non-null object
deadline_clean       478 non-null object
location             478 non-null object
scrape_date          478 non-null object
state                478 non-null object
dtypes: int64(3), object(9)
memory usage: 44.9+ KB
```

The starting and outcomes data sets were then merged to produce a master data set, which only included 478 campaigns.

Additional cleaning including using regex to break out the nested columns, specifically “creator”, “category”, and “location” to isolate the creator name, category and country, city, state.

Feature Engineering:

In order to prepare the datasets I needed to perform additional feature engineering on the text columns. I decided to leverage Pipeline, Function Transformer, and Feature Union from sklearn to streamline feature engineering and modeling.

To summarize how they’re used:

- **Pipelines**: A pipeline “a pipeline bundles preprocessing and modeling steps so you can use the whole bundle as if it were a single step.”
- **Function Transformer:** Takes a python processing job and turns it into an object that a sci-kit learn Pipeline can understand. Using Function Transformer I wrote three lambda

functions to select the text, numeric, and categorical features in order to create separate Pipelines for each.

- **Feature Union:** Allows us to join the arrays generated by the pipelines as a single array that will be the input to our classifier.

In [20]:
```
# Goal of FunctionTransformer: takes a Python job and turn it into an object that the scikit-learn Pipeline can understand
# Write 3 functions for pipeline preprocessing
# 1) Takes entire DataFrame and returns numeric columns
# 2) Takes entire DataFrame and returns text columns
# 3) Takes entire DataFrame and returns categorical columns
# Using these function transformer objects, we can build a separate Pipeline for our numeric data, text data, and categorical

#Parameter: validate = False => tells scikit-learn it doesn't need to check for NaNs or va;idate the dtypes of the input

from sklearn.base import TransformerMixin
from sklearn.pipeline import Pipeline

from sklearn.preprocessing import FunctionTransformer
from sklearn.pipeline import FeatureUnion

get_text_data = FunctionTransformer(lambda x: x['combined'], validate = False)

get_numeric_data = FunctionTransformer(lambda x: x[['goal','time_to_launch', 'launch_to_deadline', 'launched_month','launched_

# Took out creator_clean
get_categorical_data = FunctionTransformer(lambda x: x[['category_clean', 'location_clean_country','location_clean_name', 'loc
```

Once the columns were selected, I then added the necessary feature engineering steps required for each type of data (numeric, text, categorical).

For the first iteration of the model the numeric pipeline included SimpleImputer where nan's were filled with 0 in order to flag missing data. The text pipeline included CountVectorizer() to create a Bag of Words representation and the categorical pipe included an imputer with the fill value = 'No Value' and OneHotEncoder (handle unknown set to 'ignore' in order to account for the case where features were present in the test but not the training dataset).

The pipelines were further bundled into a single master pipeline that included instantiation of a LogisticRegression() object (for example).

```
# Feature Union allows us to place the arrays generated by the pipelines together
# as a single array that will be the input to our classifier

from sklearn.pipeline import FeatureUnion
from sklearn.preprocessing import OneHotEncoder
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
import numpy as np
from sklearn.impute import SimpleImputer

#get_numeric_data created in prior block as FunctionTransformers
numeric_pipeline = Pipeline([('selector', get_numeric_data),
                             ('imputer', SimpleImputer(missing_values=np.nan, strategy='constant',fill_value =0))
                            ])

text_pipeline = Pipeline([('selector',get_text_data),
                          ('vectorizer',CountVectorizer())])

categorical_pipeline = Pipeline([('selector',get_categorical_data),
                                 ('imputer', SimpleImputer(missing_values=np.nan, strategy='constant',fill_value='No Value')),
                                 ('onehot',OneHotEncoder(handle_unknown='ignore'))])
# handle_unknown: https://medium.com/hugo-ferreiras-blog/dealing-with-categorical-features-in-machine-learning-1bb70f07262d
# https://github.com/scikit-learn/scikit-learn/issues/12494

pl = Pipeline([
    ('union',FeatureUnion([
        ('numeric', numeric_pipeline),
        ('text',text_pipeline),
        ('categorical',categorical_pipeline)
    ])),
    ('logreg',LogisticRegression())
])
```

Feature Selection:

Given the short list of features and small data size I decided to use all features available and engineered in order to improve model prediction.

After creating the pipelines I then created a train-test split that allows us to fit the pipeline object and score. Using Pipeline ensures all feature engineering and modeling steps are taken care of and results in very clean code.

Model Selection & Performance:

Model choice in this case was driven by the need to handle the sparse matrix produced by the processing of the text columns. Especially given the small data sample size and even smaller amount of text, we can have many words represented very few times resulting in a very wide table.

Using a random forest classifier and a linear svc seemed to make sense as both tend to perform well in high dimensional space and in cases where the number of dimensions is greater than the number of records.

After trying a number of combinations, it seems that while some models performed slightly better than others, for the most part the level of signal I could pull from the data set sat around 65%.

Summary of approaches used and performance:

Model Type	Bag of Words	TFIDF	N-grams	Hyperparameter Tuning
Logistic Regression	Model Variant 1: Accuracy: 65%			
	Model Variant 3: Accuracy: 66%			
Linear SVC	Model Variant 2: Accuracy: 64%			
Random Forest Classifier		Model Variant 4: Accuracy: 66%		Hyperparamter tuning: Accuracy: 68%

Four model variants were explored using various NLP techniques.

- Model variant #1 utilized a logistic regression model and a Bag of Words approach to the text columns using CountVectorizer() from sklearn.
- Model variant #2 utilized a linear svc model with Bag of Words.
- Model variant #3 utilized a logistic regression with Bag of Words & TFIDF.
- Model variant #4 utilized a random forest classifier with bi-grams and TFIDF. I also performed hyperparameter tuning on the random forest classifier, which resulted in a 2% lift.

Take-Aways:

After completing the project, a couple key take-aways emerged, especially around applying NLP to text data.

1) For text data to be useful, there needs to be a lot of it. Specifically, even a few sentences per campaign produced a very wide and sparse table which, while expected in NLP

problems, resulted in the text data contributing very little in terms of lift for the base performance.

2) Spend less time on data engineering and try to find data produced in shorter intervals. The monthly scrapes combined with the average campaign only being a month long resulted in a large portion of the data needing to be discarded because of potentially biasing any models to be overly accurate due to leakage.
3) Pipelines are great but can cause downstream issues when attempting to unpack interpretability of results from the machine learning models.

However given enough data (especially around the text based features) could have yielded interesting insights into how campaign owners can craft effective messages in order to be successful in their campaigns. Ideally what I would do (time-permitting) is write a scraper to create weekly snapshots of Kickstarter campaigns and include more attributes.