Contents

DonorsChoose Application Screening

Team: *Yuwen Yan, Kexin Liang, Lin Xu, Yuxuan Wang, Maya Carnie, Pavan*

1 Project Definition & Introduction

Donorschoose.org is a non-profit crowdfunding platform that allows individuals to donate much-needed materials for public school projects. Every year, teachers submit hundreds of thousands of project proposals which are then manually screened by volunteers before posting on the website. Since they expect to receive close to 500,000 proposals in 2019, they were looking to scale the processes and improve efficiency.

The goal of the project is to predict whether a project proposal can be auto-approved so that volunteers can focus their energy on more detailed and nuanced projects. This helps in removing bias between volunteers and allows teachers to get their projects funded more quickly. By extracting the useful text features and important metadata, we should be able to provide targeted suggestions to teachers to improve their project approval and conversion rates.

2 Solution Overview

DonorsChoose currently has 500,000 projects that need to be reviewed and this number is increasing year by year. There are three main problems donors are facing.

- Improve and optimize current manual processes and resources
- Increase the consistency of project vetting across different volunteers
- Focus volunteer time on the applications that need the most assistance

Our final model will predict if a particular project can be auto-approved (binary classification) based on the numerical and text features we engineered. This information could help DonorsChoose save hundreds of thousands of time and labor. Appropriate data formats will remove biases in the model fit. The power of machine learning can help expose truth inside messy data sets, so it's possible for algorithms to help remove human bias DonorsChoose screening process bias. Besides, our model will provide immediate results to teachers. Volunteers could focus their time on applications that need the most assistance because non-approved projects will then be sent for further manual evaluation.

3 Technical Specifications

Platform	Google Colab
Runtime Language	Python 3
Hardware Accelerator	GPU

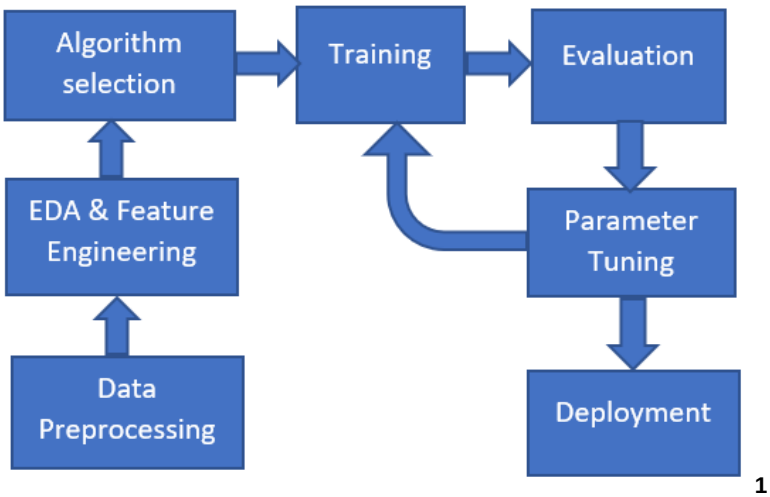
model name	Intel(R) Xeon(R) CPU @ 2.00GHz
cache size	39424 KB
address sizes	46 bits physical, 48 bits virtual

Text feature libraries	**Modeling Libraries**
NLTK - (Tokenization, Lemmatization, Stop words removal	Keras - Bidirectional RNN, Early Stop, Cross Validation
TextBlob - (Sentiment polarity)	Sklearn - Standard scaling, roc_auc_score

4 Methodology

Our approach was to first generate and choose the valuable features from the original features in the datasets. We did data transformation on numeric and categorical features and text mining on the text features. Finally, in our numeric features we have statistics based on the price and quantities of resource items, time factors, and sentimental scores for each essay, etc.
We explore different methods to encode text as numerical representations. For example, by doing word embedding, we are able to obtain context and meaning related information from the raw text data.

After feature engineering, we are left with word vectors and individual level features which are then fit into different models. We train 3 models which are Bidirectional Recurrent Neural Network, LightGBM, and XGBoost, and then compare the performance to choose the optimal one. Since this is a binary classification problem, we used appropriate loss function and activation functions for the RNN model. For LightGBM and XGBoost, we set a cross-validation to adjust the hyper-parameters iteratively. The model evaluation metric being used is AUC (Area under the ROC curve) as defined by kaggle. Finally, we use the trained model to make predictions for the test data and get the probabilities of project proposal approval.



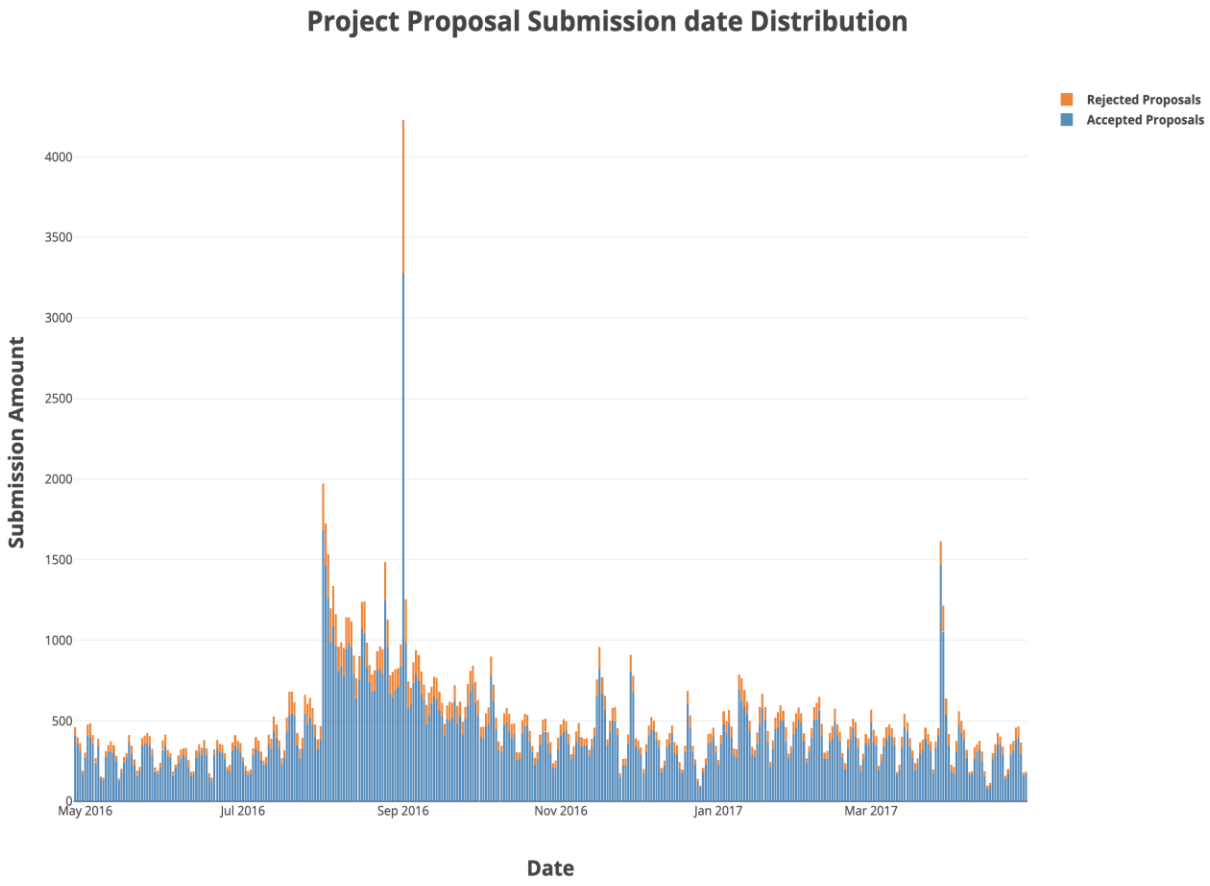
[1]

5 EDA

To have a basic overview of the whole datasets and determine which features do have impacts on the target variable, we conduct some exploratory analysis based on some particular features before going to the modeling part, helping us better understand the datasets.

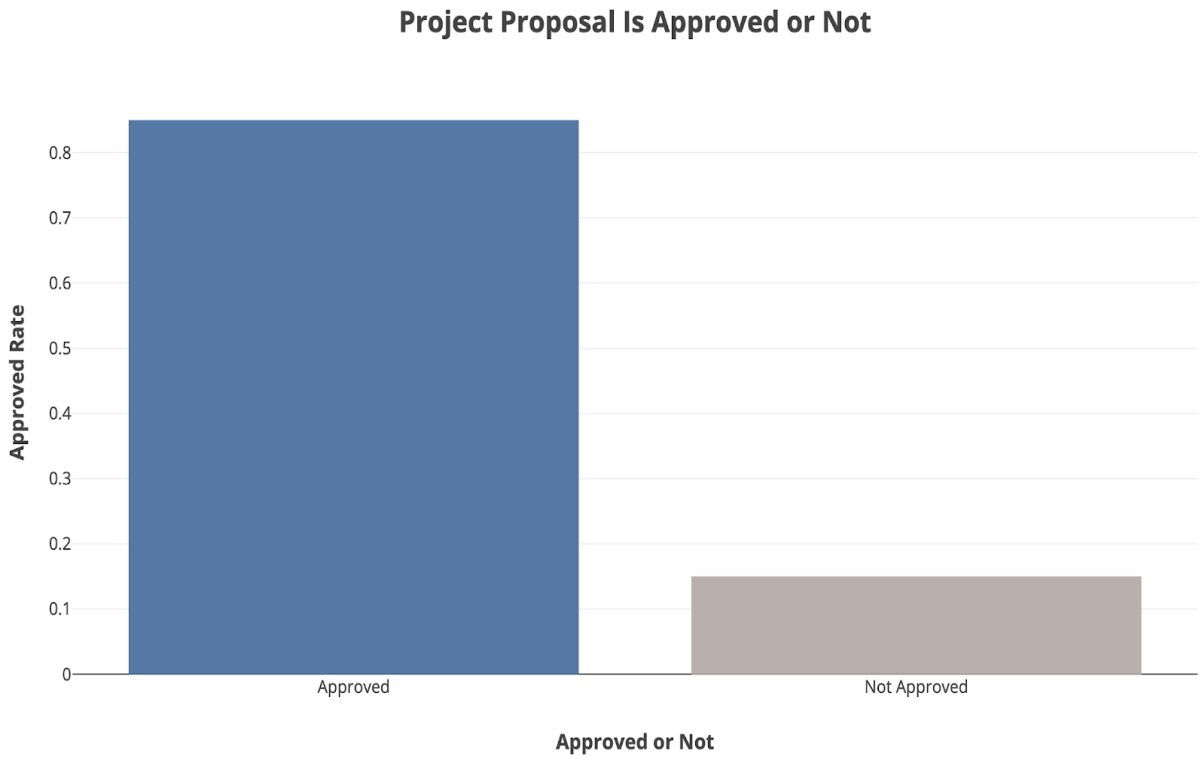
[1] https://www.sqlrelease.com/building-first-machine-learning-model-using-logistic-regression-in-python-step-by-step

5.1 Time Range of the Data



Our training data covers nearly a full year, from 4/27/2016 to 4/30/2017. There is a considerable long-term trend of numbers increasing from Jul to Sep and then falling back to a baseline toward Nov. This time range coincides with the start of the school year, for which we would expect larger numbers of proposals. This impression is confirmed by the monthly barplot in the upper right panel, where sees the numbers peaking in Aug and Sep. Toward the typical holiday period of Jun, the numbers decrease. Here the 1-yr coverage becomes useful; since we don't need to correct for some months having more data than others.

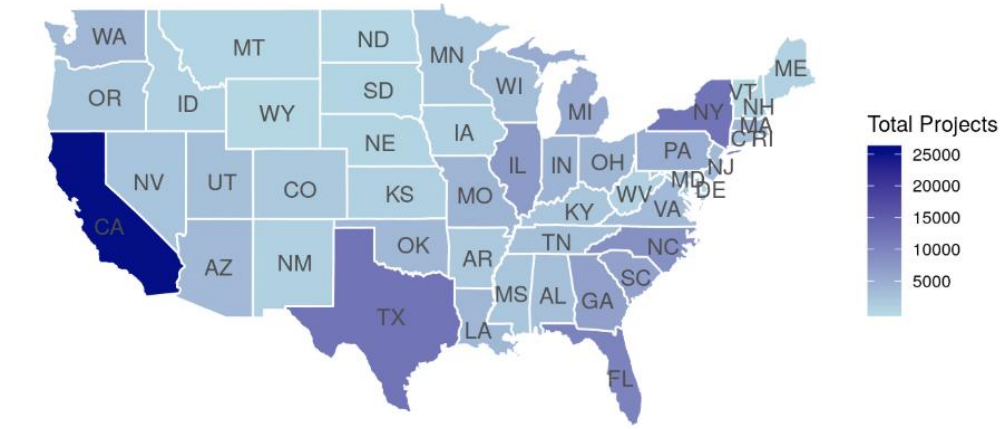
5.2 Proportion of the Target Variable (Approved or Not)



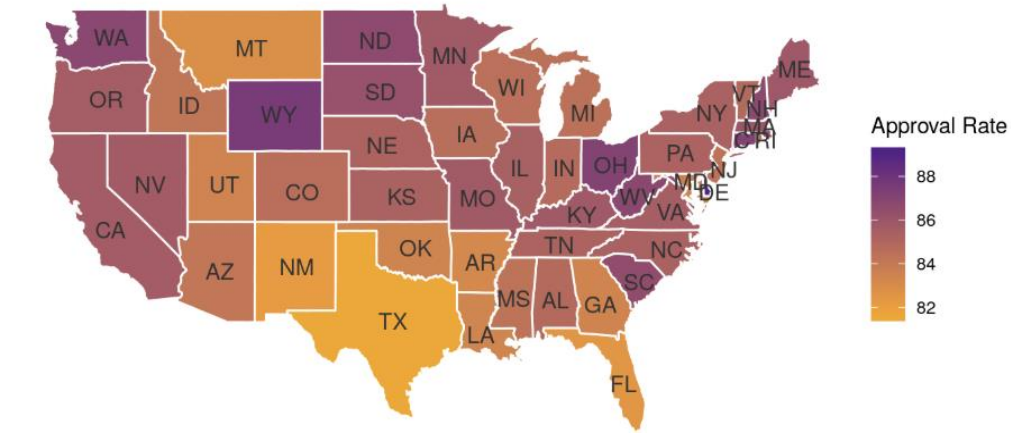
This is a pretty imbalanced dataset with approximately 85% of the proposals getting approved. From the business perspective, this is good because it shows that there were lots of interesting projects approved and supported by the DonorChoose.org to help students and teachers. Also, we need to be careful when dealing with this imbalanced dataset because it will affect how we choose hyper-parameter in cross-validation and choose evaluation metrics later.

5.3 Submission Amount and Approved Rate across States

Number of Submitted Project Proposals per US state

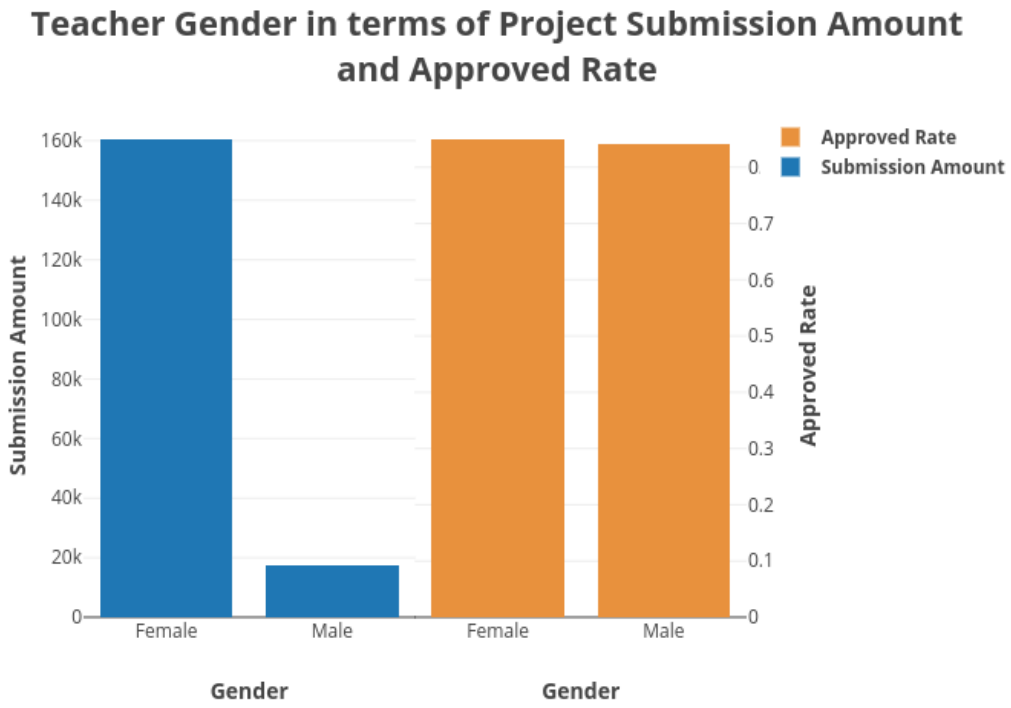


Project Approval Fraction per US state



By comparing these two graphs, we can find that the distribution of submission amount and distribution of approved rate have different patterns across states. This means that the School_State feature is impactful on the target variable. For example, CA is the state having most project proposals submitted, but WY is the state having the highest approved rate of proposals. Also, we can also find that there are no strong relations between the approval rates in neighbouring states.

5.4 Submission Amount and Approved Rate across Gender



We generate the Gender column based on the Teacher_prefix column in the original dataset. There are nearly 90% of the teachers using DonorChoose.org are female, which is a huge imbalance. But when we look at the approved rate, the rate for *female vs male* teachers are very close, which shows that there is probably no real effect present. So we won't consider the gender factor in our further analysis and modeling.

6 Data Transformation

6.1 Numeric Features

6.1.1 Statistics based on Prices and Quantities of Resource Items Required by the Proposal

Since there are duplicated proposal IDs in the original resource dataset because one proposal can require multiple kinds of resource items, we group by the proposal ID and do some aggregation on Price and Quantity features, which is used for merging with the training dataset and test dataset in the further analysis. In detail, We generate the statistics including sum, minimum, maximum, mean, and standard deviation based on the Quantity and Price respectively. Also, we calculate the average resource price for each proposal by using the

overall spending on all different resource items divided by the total quantity of different items in that proposal.

```
# Merge with resources
res = pd.DataFrame(res[['id', 'quantity', 'price']].groupby('id').agg(\
    {
        'quantity': [
            'sum',
            'min',
            'max',
            'mean',
            'std',
            # lambda x: len(np.unique(x)),
        ],
        'price': [
            'count',
            'sum',
            'min',
            'max',
            'mean',
            'std',
            lambda x: len(np.unique(x)),
        ]}
    )).reset_index()
res.columns = ['_'.join(col) for col in res.columns]
res.rename(columns={'id_': 'id'}, inplace=True)
res['mean_price'] = res['price_sum']/res['quantity_sum']
# res['price_max_to_price_min'] = res['price_max']/res['price_min']
# res['quantity_max_to_quantity_min'] = res['quantity_max']/res['quantity_min']

print(res.head())
train = train.merge(res, on='id', how='left')
test = test.merge(res, on='id', how='left')
```

6.1.2 Previous Number of Submitted Proposals and Approved Rate of the Teacher

Considering that teacher's previous experience of writing proposals and the approved rate of the proposals submitted by this teacher previously will affect whether this teachers' new proposal will be approved or not, we generate the specific columns to cover this information. We group the data by teacher ID and calculate a cumulative sum of both submitted proposals and approved proposals based on the proposal submitted time for one particular teacher. Also, we set the default approved rate as 0.5 for those who are new to Dornoschoose.org since this is a binary classification problem.

```
#Accepted projects counter (gave imrovement on CV but worse on LB, need to be implemented withing a CV loop with splitting data by time)
df_all['project_is_approved'].fillna(0, inplace=True)
cumsums = df_all[
            ['id',
            'teacher_id',
            'project_submitted_datetime',
            'project_is_approved']].\
        sort_values('project_submitted_datetime').\
        groupby(['teacher_id']).agg({'project_is_approved': lambda x: x.shift().sum(), 'id': 'first'}).fillna(0).\
        groupby(level=0).agg({'project_is_approved': 'cumsum', 'id': 'first'}).reset_index()
cumsums = pd.DataFrame(cumsums)
cumsums.rename(columns={'project_is_approved': 'teacher_number_of_previously_accepted_projects'}, inplace=True)
print(cumsums.head())
train = train.merge(cumsums, on=['id', 'teacher_id'], how='left')
test = test.merge(cumsums, on=['id', 'teacher_id'], how='left')

train['approve_rate'] = (train['teacher_number_of_previously_accepted_projects'] + 5)/\
    (train['teacher_number_of_previously_posted_projects'] + 10)
test['approve_rate'] = (test['teacher_number_of_previously_accepted_projects'] + 5)/\
    (test['teacher_number_of_previously_posted_projects'] + 10)
```

6.1.3 Time Factors Extracted from the Original Submitted Datetime Feature

Since we have the original submission time feature which cannot be fit into model directly using the original timestamp format, we simply extract different time factors from this original feature. The main time factors extracted are year, month, date, day of the week, hour and minute.

```
def process_timestamp(df):
    df['year'] = df['project_submitted_datetime'].apply(lambda x: int(x.split('-')[0]))
    df['month'] = df['project_submitted_datetime'].apply(lambda x: int(x.split('-')[1]))
    df['date'] = df['project_submitted_datetime'].apply(lambda x: int(x.split(' ')[0].split('-')[2]))
    df['day_of_week'] = pd.to_datetime(df['project_submitted_datetime']).dt.weekday
    df['hour'] = df['project_submitted_datetime'].apply(lambda x: int(x.split(' ')[-1].split(':')[0]))
    df['minute'] = df['project_submitted_datetime'].apply(lambda x: int(x.split(' ')[-1].split(':')[1]))
    df['project_submitted_datetime'] = pd.to_datetime(df['project_submitted_datetime']).values.astype(np.int64)

process_timestamp(train)
process_timestamp(test)
```

6.1.4 Length and Word Count of Text Features

Besides the transformation of original numeric features, we also generate numeric variables such as length of characters and word count for the text features. We did this manipulation for proposal title, essay, and resource summary respectively.

```
# Extract features
def extract_features(df):
    df['project_title_len'] = df['project_title'].apply(lambda x: len(str(x)))
    df['project_essay_1_len'] = df['project_essay_1'].apply(lambda x: len(str(x)))
    df['project_essay_2_len'] = df['project_essay_2'].apply(lambda x: len(str(x)))
    df['project_essay_3_len'] = df['project_essay_3'].apply(lambda x: len(str(x)))
    df['project_essay_4_len'] = df['project_essay_4'].apply(lambda x: len(str(x)))
    df['project_resource_summary_len'] = df['project_resource_summary'].apply(lambda x: len(str(x)))

    df['project_title_wc'] = df['project_title'].apply(lambda x: len(str(x).split(' ')))
    df['project_essay_1_wc'] = df['project_essay_1'].apply(lambda x: len(str(x).split(' ')))
    df['project_essay_2_wc'] = df['project_essay_2'].apply(lambda x: len(str(x).split(' ')))
    df['project_essay_3_wc'] = df['project_essay_3'].apply(lambda x: len(str(x).split(' ')))
    df['project_essay_4_wc'] = df['project_essay_4'].apply(lambda x: len(str(x).split(' ')))
    df['project_resource_summary_wc'] = df['project_resource_summary'].apply(lambda x: len(str(x).split(' ')))

extract_features(train)
extract_features(test)
```

6.1.5 Sentiment score

Sentiment analysis is basically the process of determining the attitude or the emotion of the writer, i.e., whether it is positive or negative or neutral.

The sentiment function of textblob returns two properties, polarity, and subjectivity. Polarity is float which lies in the range of [-1,1] where 1 means positive statement and -1 means a negative statement. Subjective sentences generally refer to personal opinion, emotion or judgment whereas objective refers to factual information. Subjectivity is also a float which lies in the range of [0,1]. We conduct sentiment analysis on both proposal title and essay and obtain each polarity and subjectivity respectively.

```
# functions to get polatiy and subjectivity
def get_polarity(text):
    textblob = TextBlob(text)
    pol = textblob.sentiment.polarity
    return round(pol,3)

def get_subjectivity(text):
    textblob = TextBlob(text)
    subj = textblob.sentiment.subjectivity
    return round(subj,3)
X_train1['title_polarity'] = X_train1['project_title'].apply(get_polarity)
X_train1['title_subjectivity'] = X_train1['project_title'].apply(get_subjectivity)
X_train1['essay_1_polarity'] = X_train1['project_essay_1'].apply(get_polarity)
X_train1['essay_1_subjectivity'] = X_train1['project_essay_1'].apply(get_subjectivity)
X_train1['essay_2_polarity'] = X_train1['project_essay_2'].apply(get_polarity)
X_train1['essay_2_subjectivity'] = X_train1['project_essay_2'].apply(get_subjectivity)
```

6.2 Categorical Features

For categorical features, we keep most of them except the teacher gender since we proved that gender doesn't have impact on the target variable. So we keep School state, Project grade category, Project subject categories and Project subject subcategories in the dataset, and conduct a label encoder on these categorical features to convert the labels into numeric form so as to convert it into the machine-readable form. Basically, it assigns a single unique number to each label in one feature.

```
# Preprocess columns with label encoder
print('Label Encoder...')
cols = [
    'teacher_id',
    'teacher_prefix',
    'school_state',
    'project_grade_category',
    'project_subject_categories',
    'project_subject_subcategories'
]

for c in tqdm(cols):
    le = LabelEncoder()
    le.fit(df_all[c].astype(str))
    train[c] = le.transform(train[c].astype(str))
    test[c] = le.transform(test[c].astype(str))
del le
gc.collect()
print('Done.')
```

6.3 Text Feature

6.3.1 Data Overview

For each proposal, besides project resource summary, we can extract the following text features:

- Project_title: a free-form text feature with probably only a word limit constraint.
- Project_essay :
    - **Before 2016/5/17**
        - Project_essay_1: classroom introduction (what challenge the students faced)
        - Project_essay_2: students description
        - Project_essay_3: how the students will use the resources requested
        - Project_essay_4: why the project is important
    - **After 2016/5/17**
        - Project_essay_1: Describe your students - What makes your students special? Specific details about their background, your neighborhood, and your school are all helpful. (similar to original essay 1 and 2)
        - Project_essay_2: About your project: How will these materials make a difference in your students' learning and improve their school lives? (similar to original essay 3 and 4)
        - Project_essay_3: NA
        - Project_essay_4: NA

Since those columns mostly provide related information, our first step is to combine all these columns into one text feature named “application_text”, then we replace all null values with `NAN_WORD = "_NAN_".`

```
X_train["application_text"] = X_train["project_title"].map(str) +\
                        X_train["project_essay_1"].map(str) +\
                        X_train["project_essay_2"].map(str) + \
                        X_train["project_essay_3"].map(str) + \
                        X_train["project_essay_4"].map(str)
X_train["application_text"] = X_train["application_text"].fillna(NAN_WORD)
```

6.3.2 Text Feature Pre-processing

- Text decontraction

In this step, we expand English language contractions, for example, transform “won’t” to “will not”.

```
## decontracted
import re

def decontracted(phrase):
    # specific
    phrase = re.sub(r"won't", "will not", phrase)
    phrase = re.sub(r"can\'t", "can not", phrase)

    # general
    phrase = re.sub(r"n\'t", " not", phrase)
    phrase = re.sub(r"\'re", " are", phrase)
    phrase = re.sub(r"\'s", " is", phrase)
    phrase = re.sub(r"\'d", " would", phrase)
    phrase = re.sub(r"\'ll", " will", phrase)
    phrase = re.sub(r"\'t", " not", phrase)
    phrase = re.sub(r"\'ve", " have", phrase)
    phrase = re.sub(r"\'m", " am", phrase)
    return phrase
```

- Removing stop-words and punctuations:

We define a function to takes in a string of text, then performs the following:
1. Remove all punctuation
2. Remove all stopwords such as 'i', 'me', 'the'
3. Returns a list of the cleaned text

```
from nltk.corpus import stopwords
nltk.download("stopwords")


def text_process(mess):

    # Check characters to see if they are in punctuation
    nopunc = [char for char in mess if char not in string.punctuation]

    # Join the characters again to form the string.
    nopunc = ''.join(nopunc)

    # Now just remove any stopwords
    return [word for word in nopunc.split() if word.lower() not in stopwords.words('english')]
```

- Remove useless tags like '\\r' and '\\"'

```
preprocessed_essays_X_test = []
# tqdm is for printing the status bar
for sentance in tqdm(X_test["application_text"]):
    sent = decontracted(sentance)
    sent = sent.replace('\\r', ' ')
    sent = sent.replace('\\"', ' ')
    sent = sent.replace('\\n', ' ')
    sent = re.sub('[^A-Za-z0-9]+', ' ', sent)
    # https://gist.github.com/sebleier/554280
    sent = ' '.join(e for e in sent.split() if e.lower() not in stopwords)
    preprocessed_essays_X_test.append(sent.lower().strip())
```

- Tokenization

We define a tokenization function to convert sentences we obtain from previous steps to words. The function return tokenized sentences which include a list of token id for each application, and a word dictionary where map each word to a token id.

```
## tokenize
import nltk
nltk.download("punkt")
def tokenize_sentences(sentences, words_dict):
    tokenized_sentences = []
    for sentence in tqdm(sentences):
        if hasattr(sentence, "decode"):
            sentence = sentence.decode("utf-8")
        tokens = nltk.tokenize.word_tokenize(sentence)
        result = []
        for word in tokens:
            word = word.lower()
            if word not in words_dict:
                words_dict[word] = len(words_dict)
            word_index = words_dict[word]
            result.append(word_index)
        tokenized_sentences.append(result)
    return tokenized_sentences, words_dict
```

```
print("Loading data...")
list_sentences_X_train = np.asarray(preprocessed_essays_X_train)
list_sentences_X_test = np.asarray(preprocessed_essays_X_test)
print("Done.")
print("Tokenizing sentences in train set...")
tokenized_sentences_train, words_dict = tokenize_sentences(list_sentences_X_train, {})
print("Tokenizing sentences in test set...")
tokenized_sentences_test, words_dict = tokenize_sentences(list_sentences_X_test, words_dict)
```

After text data preprocessing, for each application essay, we extract a refined list of words represented as tokenid.

6.3.3 Text Vectorization

In order to perform machine learning on text, we need to transform our documents into vector representations such that we can apply numeric machine learning. This is an essential first step toward language-aware analysis. There are many methods that can vectorize text data, we explore three of them：

6.3.3.1 TF-IDF

TF-IDF means term frequency-inverse document frequency, and it is a scoring scheme for words that measures how important a word is to a document. TF-IDF believes that high frequency may not be able to provide much information gain and gives importance to terms that occur in a few documents. In other words, rare words contribute more weight. We choose TF-IDF first mainly because of 2 reasons:

- Building a baseline model. By using scikit-learn, with only a few lines of code we are able to perform vectorization. Later on, we can Deep Learning to bit it.
- In our case, TF-IDF might not be inferior to word embedding because sometimes frequently occurring words are actually strongly indicative of the task we are trying to solve.

Also, we should remember TF-IDF is based on the bag-of-words (BoW) model, therefore it does not capture position in the text, semantics, co-occurrences in different documents, etc.

```
from sklearn.feature_extraction.text import TfidfVectorizer
vectorizer = TfidfVectorizer(min_df=10)
text_tfidf = vectorizer.fit_transform(preprocessed_essays_X_train)
print("Shape of matrix after one hot encodig ",text_tfidf.shape)
```

```
Shape of matrix after one hot encodig  (182080, 21570)
```

The TfidfVectorizer will tokenize documents, learn the vocabulary and inverse document frequency weightings, and allow us to encode new documents. A vocabulary is learned from the documents and each word is assigned a unique integer index in the output vector.The inverse document frequencies are calculated for each word in the vocabulary, assigning the lowest

score of 1.0 to the most frequently observed word. Finally, each document is encoded as a sparse array and the scores are normalized to values between 0 and 1 and the encoded document vectors can then be used directly with most machine learning algorithms.

```
from sklearn.decomposition import TruncatedSVD
svd = TruncatedSVD(n_components=320, random_state=50)
textidf = svd.fit_transform(text_tfidf)
```

From above we can observe the matrix after tf-idf vectorization is very sparse and make it hard for us to process, so we use singular value decomposition (SVD) dimensionality reduction techniques to help the feature extraction by using logarithm TF-IDF method.

The performance might not be good because we limit the n_componetents to 320 which removes much information from original data. By comparison we also try not to join text features, instead, we process proposal essay, proposal title, and resources description and obtain separately.

```
# Preprocess text
print('Preprocessing text...')
cols = [
    'project_title',
    'project_essay',
    'project_resource_summary'
]
n_features = [
    400,
    4040,
    400,
]

for c_i, c in tqdm(enumerate(cols)):
    tfidf = TfidfVectorizer(
        max_features=n_features[c_i],
        norm='l2',
        )
    tfidf.fit(df_all[c])
    tfidf_train = np.array(tfidf.transform(train[c]).toarray(), dtype=np.float16)
    tfidf_test = np.array(tfidf.transform(test[c]).toarray(), dtype=np.float16)

    for i in range(n_features[c_i]):
        train[c + '_tfidf_' + str(i)] = tfidf_train[:, i]
        test[c + '_tfidf_' + str(i)] = tfidf_test[:, i]
```

6.3.3.2 Word Embedding

Word embedding represents words and phrases in vectors of (non-binary) numeric values with much lower and thus denser dimensions. An intuitive assumption for good word embedding is that they can approximate the similarity between words. It can be learned using a variety of language models. Instead of training our own model, we use pre-trained word vectors derived from the two most common method word2vec and glove to vectorize our text feature.

6.3.3.2.1 Word2vec

Word2vec is great for going deeper into the documents we have and helps in identifying content and subsets of content. Its vectors represent each word's context. (i.e the n-gram of which it is a part). Word2vec can be seen as a model that improves its ability to predict [(target word | context words). The learned vectors are fed into a discriminative model (typically an RNN). First, we define a function to read a pretrained vector file and obtain an embedding list and embedding word dictionary.

```
#embedding

def read_embedding_list(file_path):
    embedding_word_dict = {}
    embedding_list = []
    f = open(file_path, encoding="utf8")
    #f = open(file_path)
    for index, line in enumerate(f):
        if index == 0:
            continue
        values = line.split()
        word = values[0]
        try:
            coefs = np.asarray(values[1:], dtype='float32')
        except:
            continue
        embedding_list.append(coefs)
        embedding_word_dict[word] = len(embedding_word_dict)
    f.close()
    embedding_list = np.array(embedding_list)
    return embedding_list, embedding_word_dict
```

```
# Embedding

embedding_path = "/content/gdrive/Shared drives/Predictive Analytics Project/Feature Engineering/crawl-300d

print("Loading embeddings...")
embedding_list, embedding_word_dict = read_embedding_list(embedding_path)
embedding_size = len(embedding_list[0])
print("Done")
```

The embedding matrix here will be used as the embedding layer in neural network.

```
def convert_tokens_to_ids(tokenized_sentences, words_list, embedding_word_dict, sentences_length):
    words_train = []

    for sentence in tokenized_sentences:
        current_words = []
        for word_index in sentence:
            word = words_list[word_index]
            word_id = embedding_word_dict.get(word, len(embedding_word_dict) - 2)
            current_words.append(word_id)

        if len(current_words) >= sentences_length:
            current_words = current_words[:sentences_length]
        else:
            current_words += [len(embedding_word_dict) - 1] * (sentences_length - len(current_words))
        words_train.append(current_words)
    return words_train
```

```
print("Preparing data...")
embedding_list, embedding_word_dict = clear_embedding_list(embedding_list, embedding_word_dict, words_dict)

embedding_word_dict[UNKNOWN_WORD] = len(embedding_word_dict)
embedding_list.append([0.] * embedding_size)
embedding_word_dict[END_WORD] = len(embedding_word_dict)
embedding_list.append([-1.] * embedding_size)

embedding_matrix = np.array(embedding_list)

id_to_word = dict((id, word) for word, id in words_dict.items())
train_list_of_token_ids = convert_tokens_to_ids(
    tokenized_sentences_train,
    id_to_word,
    embedding_word_dict,
    sentences_length)
test_list_of_token_ids = convert_tokens_to_ids(
    tokenized_sentences_test,
    id_to_word,
    embedding_word_dict,
    sentences_length)
X_train_text = np.array(train_list_of_token_ids)
X_test_text = np.array(test_list_of_token_ids)
print("Done")
```

6.3.3.2.2 GloVe

GloVe means a global vector for word representation and it is another model of distributed word representation. Word2Vec does not take advantage of global context because both CBOW and Skip-Grams in Word2vec model are “predictive” models, in that they only take local contexts into account. GloVe embeddings by contrast leverage the same intuition behind the cooccurrence matrix used distributional embeddings. The usage of Glove pre-trained vector is pretty much the same as word2vec, therefore, we only make some exploration here and didn’t feed this version of vector into models due to limited resources.

```
def loadGloveModel(gloveFile):

    print ("Loading Glove Model")

    f = open(gloveFile,'r', encoding = 'utf8')

    model = {}

    for line in tqdm(f):
        splitLine = line.split()
        word = splitLine[0]
        embedding = np.array([float(val) for val in splitLine[1:]])
        model[word] = embedding

    print ("Done.",len(model)," words loaded!")

    return model
```

```
model = loadGloveModel('/Predictive Analytics Project/Feature Engineering/glove.42B.300d.txt')
```

```
Loading Glove Model
HBox(children=(IntProgress(value=1, bar_style='info', max=1), HTML(value='')))

Done. 1917494  words loaded!
```

After the loading glove model, we can use GLoVe vector to transform each application review into a vector as a length of 300.

```
import pickle
with open('glove.42B.300d.txt', 'wb') as f:
    pickle.dump(words_corpus, f)

with open('glove.42B.300d.txt', 'rb') as f:
    model = pickle.load(f)
    glove_words =  set(model.keys())

# average Word2Vec
# compute average word2vec for each review.
avg_w2v_vectors = []; # the avg-w2v for each sentence/review is stored in this list
for sentence in tqdm(preprocessed_essays): # for each review/sentence
    vector = np.zeros(300) # as word vectors are of zero length
    cnt_words =0; # num of words with a valid vector in the sentence/review
    for word in sentence.split(): # for each word in a review/sentence
        if word in glove_words:
            vector += model[word]
            cnt_words += 1
    if cnt_words != 0:
        vector /= cnt_words
    avg_w2v_vectors.append(vector)

print(len(avg_w2v_vectors))
print(len(avg_w2v_vectors[0]))
```

7 Modeling Analysis

This is essentially a probabilistic classification problem and the main objective is to predict whether or not a project proposal submitted by a teacher will be approved. Three models are proposed to Donorschoose.org – XGBoost, LightGBM, and Bidirectional Recurrent Neural Network.

7.1 Train-Validation-Test Split

Kaggle provides train and test data for performance evaluation, but the ground truth of test data is unknown for us. Therefore, we split the train data into train(80%) and validation(20%) before training the model. Validation data can help detect the model performance on our local machine, avoid overfitting and set hyperparameters.

7.2 Model Selection

For model selection, we started with the relatively simple tree-based model, XGBoost. Tree boosting has been shown to provide great results on many standard classification benchmarks.

7.2.1XGBoost

XGBoost is a decision-tree-based ensemble algorithm that uses a gradient boosting framework. An important factor behind the success of XGBoost is its scalability in all scenarios. Additionally, XGBoost is robust enough to support fine-tuning and regularization parameters.

K-Fold Cross Validation was used to divide the data into 5 folds to ensure that each fold was used during testing at some point. The data was then split into training and validation groups, which were later saved into a matrix. Next, parameters were set and the model was trained using early stopping after 30 rounds. However, XGBoost only gives a test AUC of 0.66.

```
cv_scores = []
xgb_preds = []
seed = 28 # Get your own seed

K = 5
kf = KFold(n_splits = K, random_state = seed, shuffle = True)

for train_index, test_index in kf.split(X):

    # Split out a validation set
    X_train, X_valid, y_train, y_valid = train_test_split(X, y, test_size=0.20, random_state=random.seed(seed))

    # params are tuned with kaggle kernels in mind
    xgb_params = {'eta': 0.15,
                  'max_depth': 7,
                  'subsample': 0.80,
                  'colsample_bytree': 0.80,
                  'objective': 'binary:logistic',
                  'eval_metric': 'auc',
                  'seed': seed
                 }

    d_train = xgb.DMatrix(X_train, y_train)
    d_valid = xgb.DMatrix(X_valid, y_valid)
    d_test = xgb.DMatrix(X_test)

    watchlist = [(d_train, 'train'), (d_valid, 'valid')]
    model = xgb.train(xgb_params, d_train, 2000, watchlist, verbose_eval=50, early_stopping_rounds=30)
    cv_scores.append(float(model.attributes()['best_score']))
    xgb_pred = model.predict(d_test)
    xgb_preds.append(list(xgb_pred))

p = model.predict(d_test, num_iteration=model.best_iteration)
```

7.2.2 LightGBM

Since we only get test AUC of 0.66 on XGBoost. Then we build our second model using LightGBM and finding AUC(Area Under Curve). LightGBM is also a gradient boosting framework that uses tree-based learning algorithm. LightGBM grows tree vertically while other algorithms grows trees horizontally. It will choose the leaf with max delta loss to grow. When

growing the same leaf, the Leaf-wise algorithm can reduce more loss than a level-wise algorithm.[2] Compared to XGBoost, LightGBM can produce the same accurate predictions as XGBoost, but can be 25 times faster.

LightGBM can also need to handle categorical features by taking the input of feature names. It does not convert to one-hot coding, and we use label encoding which is much faster than one-hot coding.

We use two different language processing models to preprocess the text features in LightGBM, one with TF-IDF and the other on with Word2Vector.

7.2.2.1 LightGBM + TF- IDF

We first try TF-IDF to convert a collection of raw documents into a matrix of TF-IDF features. Comparing the results to XGBoost baseline model, the Light model performed much better than the baseline. It gives us a better AUC of 0.78 on validation dataset and 0.79 on test dataset. We used different hyper-parameters settings to tune our model. This hyper-parameter setting gives us the best result.

Code snippets are shown below:

```
# Build the model
cnt = 0
p_buf = []
n_splits = 3
n_repeats = 1
kf = RepeatedKFold(
    n_splits=n_splits,
    n_repeats=n_repeats,
    random_state=0)
auc_buf = []

for train_index, valid_index in kf.split(X):
    print('Fold {}/{}'.format(cnt + 1, n_splits))
    params = {
        'boosting_type': 'gbdt',
        'objective': 'binary',
        'metric': ['auc','acc'],
        'max_depth': 12,
        'num_leaves': 28,
        'learning_rate': 0.025,
        'feature_fraction': 0.85,
        'bagging_fraction': 0.85,
        'bagging_freq': 5,
        'verbose': 0,
        'num_threads': 1,
        'lambda_l2': 1.0,
        'min_gain_to_split': 0,
    }
```

[2] https://medium.com/@pushkarmandot/https-medium-com-pushkarmandot-what-is-lightgbm-how-to-implement-it-how-to-fine-tune-the-parameters-60347819b7fc

```
        lgb_train = lgb.Dataset(
            X.loc[train_index],
            y.loc[train_index],
            feature_name=feature_names,
            )
        lgb_train.raw_data = None

        lgb_valid = lgb.Dataset(
            X.loc[valid_index],
            y.loc[valid_index],
            )
        lgb_valid.raw_data = None

        model = lgb.train(
            params,
            lgb_train,
            num_boost_round=10000,
            valid_sets=[lgb_train, lgb_valid],
            early_stopping_rounds=100,
            verbose_eval=100,
        )

        if cnt == 0:
            importance = model.feature_importance()
            model_fnames = model.feature_name()
            tuples = sorted(zip(model_fnames, importance), key=lambda x: x[1])[::-1]
            tuples = [x for x in tuples if x[1] > 0]
            print('Important features:')
            for i in range(60):
                if i < len(tuples):
                    print(tuples[i])
                else:
                    break
```

```
        del importance, model_fnames, tuples

    p = model.predict(X.loc[valid_index], num_iteration=model.best_iteration)
    auc = roc_auc_score(y.loc[valid_index], p)


    print('{} AUC: {}'.format(cnt, auc))

    p = model.predict(X_test, num_iteration=model.best_iteration)
    if len(p_buf) == 0:
        p_buf = np.array(p, dtype=np.float16)
    else:
        p_buf += np.array(p, dtype=np.float16)
    auc_buf.append(auc)

    cnt += 1
    if cnt > 0: # Comment this to run several folds
        break

    del model, lgb_train, lgb_valid, p
    gc.collect

auc_mean = np.mean(auc_buf)
auc_std = np.std(auc_buf)
print('AUC = {:.6f} +/- {:.6f}'.format(auc_mean, auc_std))
```

We train our model until the validation score does not improve for 100 rounds.

```
Fold 1/3
Training until validation scores don't improve for 100 rounds.
[100]   training's auc: 0.766992        valid_1's auc: 0.749549
[200]   training's auc: 0.798037        valid_1's auc: 0.764917
[300]   training's auc: 0.817734        valid_1's auc: 0.771918
[400]   training's auc: 0.833036        valid_1's auc: 0.775705
[500]   training's auc: 0.845459        valid_1's auc: 0.777767
[600]   training's auc: 0.856284        valid_1's auc: 0.779143
[700]   training's auc: 0.865911        valid_1's auc: 0.780059
[800]   training's auc: 0.874811        valid_1's auc: 0.781113
[900]   training's auc: 0.882835        valid_1's auc: 0.781445
[1000]  training's auc: 0.890194        valid_1's auc: 0.781947
[1100]  training's auc: 0.897382        valid_1's auc: 0.782155
[1200]  training's auc: 0.903822        valid_1's auc: 0.782291
[1300]  training's auc: 0.910321        valid_1's auc: 0.782363
[1400]  training's auc: 0.916076        valid_1's auc: 0.782479
[1500]  training's auc: 0.921486        valid_1's auc: 0.782702
[1600]  training's auc: 0.926476        valid_1's auc: 0.78264
Early stopping, best iteration is:
[1538]  training's auc: 0.92343 valid_1's auc: 0.782815
```

7.2.2.2 LightGBM + Word2Vec

We also try Word2vec to produce word embeddings and make the model trained to reconstruct linguistic contexts of words. Word vectors are positioned in the vector space such that words that share common contexts in the corpus are located close to one another in the space. However, comparing the results of LightGBM with TF-IDF, we get a lower performance with 0.75 on the validation dataset and 0.72 on the test dataset.

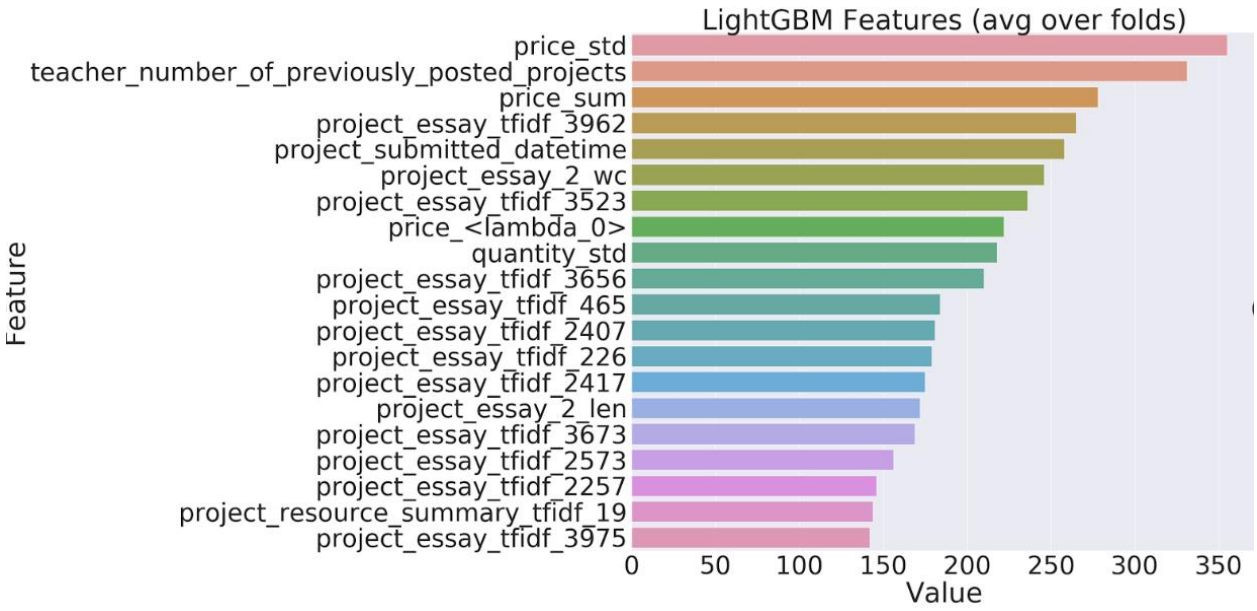
```
Training until validation scores don't improve for 100 rounds.
[100]   training's auc: 0.747794        valid_1's auc: 0.740719
[200]   training's auc: 0.756778        valid_1's auc: 0.74657
[300]   training's auc: 0.765476        valid_1's auc: 0.750668
[400]   training's auc: 0.773101        valid_1's auc: 0.753025
[500]   training's auc: 0.780001        valid_1's auc: 0.754557
[600]   training's auc: 0.786501        valid_1's auc: 0.755267
[700]   training's auc: 0.792774        valid_1's auc: 0.755738
[800]   training's auc: 0.798895        valid_1's auc: 0.756269
[900]   training's auc: 0.804662        valid_1's auc: 0.75651
[1000]  training's auc: 0.810331        valid_1's auc: 0.756826
[1100]  training's auc: 0.815825        valid_1's auc: 0.756882
[1200]  training's auc: 0.82116 valid_1's auc: 0.757132
[1300]  training's auc: 0.826263        valid_1's auc: 0.757219
[1400]  training's auc: 0.83106 valid_1's auc: 0.757265
Early stopping, best iteration is:
[1398]  training's auc: 0.830955        valid_1's auc: 0.757288
```

7.2.2.3 Feature Selection

We don't need to do feature selection before we fit that data into models. The tree-based model can be used to evaluate the importance of features that can help us generate important features with values. After the model training is completed, we create a plotImp function of the training model to obtain the importance of the features.

```
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.simplefilter(action='ignore', category=FutureWarning)

def plotImp(model, X , num = 20):
    feature_imp = pd.DataFrame({'Value':model.feature_importance(),'Feature':X.columns})
    plt.figure(figsize=(40, 20))
    sns.set(font_scale = 5)
    sns.barplot(x="Value", y="Feature", data=feature_imp.sort_values(by="Value",
                                                    ascending=False)[0:num])
    plt.title('LightGBM Features (avg over folds)')
    plt.tight_layout()
    plt.savefig('/content/gdrive/Shared drives/Predictive Analytics Project/Data/lgbm_importances-01.png')
    plt.show()
```



(Prediction power for each attribute used LightGMB with TFIDF)

7.2.3 Bidirectional Recurrent Neural Network

The last model we tried is Bidirectional Recurrent Neural Network. Bidirectional Recurrent Neural Network is the combination of two independent RNNs. The input sequence is fed in normal order for one network and in reverse order for the other network. With this form of generative deep learning, the model output can capture both the backward and forward

information about the sequence at each state, which can get more useful information from the vectorized text data.[3] Considering text data is one of the most important attributes in our model, we hypothesize the algorithm is suitable for the Donors project as bidirectional RNN can better capture the contextual data for each word.

```
gru_len = 128
Routings = 5
Num_capsule = 10
Dim_capsule = 16
dropout_p = 0.3
rate_drop_dense = 0.3

def squash(x, axis=-1):
    s_squared_norm = K.sum(K.square(x), axis, keepdims=True)
    scale = K.sqrt(s_squared_norm + K.epsilon())
    return x / scale
```

```
# capsule layer

class Capsule(Layer):
    def __init__(self, num_capsule, dim_capsule, routings=3, kernel_size=(9, 1), share_weights=True,
                 activation='default', **kwargs):
        super(Capsule, self).__init__(**kwargs)
        self.num_capsule = num_capsule
        self.dim_capsule = dim_capsule
        self.routings = routings
        self.kernel_size = kernel_size
        self.share_weights = share_weights
        if activation == 'default':
            self.activation = squash
        else:
            self.activation = Activation(activation)

    def build(self, input_shape):
        super(Capsule, self).build(input_shape)
        input_dim_capsule = input_shape[-1]
        if self.share_weights:
            self.W = self.add_weight(name='capsule_kernel',
                                     shape=(1, input_dim_capsule,
                                            self.num_capsule * self.dim_capsule),
                                     # shape=self.kernel_size,
                                     initializer='glorot_uniform',
                                     trainable=True)
        else:
            input_num_capsule = input_shape[-2]
            self.W = self.add_weight(name='capsule_kernel',
                                     shape=(input_num_capsule,
                                            input_dim_capsule,
                                            self.num_capsule * self.dim_capsule),
                                     initializer='glorot_uniform',
                                     trainable=True)
```

[3] Understanding Bidirectional RNN in PyTorch. https://towardsdatascience.com/understanding-bidirectional-rnn-in-pytorch-5bd25a5dd66

```
    def call(self, u_vecs):
        if self.share_weights:
            u_hat_vecs = K.conv1d(u_vecs, self.W)
        else:
            u_hat_vecs = K.local_conv1d(u_vecs, self.W, [1], [1])

        batch_size = K.shape(u_vecs)[0]
        input_num_capsule = K.shape(u_vecs)[1]
        u_hat_vecs = K.reshape(u_hat_vecs, (batch_size, input_num_capsule,
                                            self.num_capsule, self.dim_capsule))
        u_hat_vecs = K.permute_dimensions(u_hat_vecs, (0, 2, 1, 3))
        # final u_hat_vecs.shape = [None, num_capsule, input_num_capsule, dim_capsule]

        b = K.zeros_like(u_hat_vecs[:, :, :, 0])  # shape = [None, num_capsule, input_num_capsule]
        for i in range(self.routings):
            b = K.permute_dimensions(b, (0, 2, 1))  # shape = [None, input_num_capsule, num_capsule]
            c = K.softmax(b)
            c = K.permute_dimensions(c, (0, 2, 1))
            b = K.permute_dimensions(b, (0, 2, 1))
            outputs = self.activation(K.batch_dot(c, u_hat_vecs, [2, 2]))
            if i < self.routings - 1:
                b = K.batch_dot(outputs, u_hat_vecs, [2, 3])

        return outputs

    def compute_output_shape(self, input_shape):
        return (None, self.num_capsule, self.dim_capsule)
```

```
# define roc_callback, inspired by https://github.com/keras-team/keras/issues/6050#issuecomment-329996505
def auc_roc(y_true, y_pred):
    # any tensorflow metric
    value, update_op = tf.contrib.metrics.streaming_auc(y_pred, y_true)

    # find all variables created for this metric
    metric_vars = [i for i in tf.local_variables() if 'auc_roc' in i.name.split('/')[1]]

    # Add metric variables to GLOBAL_VARIABLES collection.
    # They will be initialized for new session.
    for v in metric_vars:
        tf.add_to_collection(tf.GraphKeys.GLOBAL_VARIABLES, v)

    # force to update metric values
    with tf.control_dependencies([update_op]):
        value = tf.identity(value)
        return value
```

```
def get_model1(embedding_matrix, sequence_length, dropout_rate, recurrent_units, dense_size):

    nontext = Input(shape=[X_train_nontextpart.shape[1]], name="nontext")
    nontext1 = Dense(64, activation='relu')(nontext)
    nontext1 = Dense(128, activation='relu')(nontext1)

    textpart = Input(shape=(sequence_length,),name="text")

    embed_layer = Embedding(embedding_matrix.shape[0], embedding_matrix.shape[1],
                            weights=[embedding_matrix], trainable=False)(textpart)
    embed_layer = SpatialDropout1D(rate_drop_dense)(embed_layer)

    x_text = Bidirectional(
        GRU(gru_len, activation='relu', dropout=dropout_p, recurrent_dropout=dropout_p, return_sequences=True))(
        embed_layer)
    capsule = Capsule(num_capsule=Num_capsule, dim_capsule=Dim_capsule, routings=Routings,
                      share_weights=True)(x_text)
    capsule = Flatten()(capsule)
    capsule = Dropout(dropout_p)(capsule)

    X = concatenate([
        nontext1,
        capsule,
    ])

    output = Dense(1, activation='sigmoid')(X)
    model = Model(inputs=[nontext, textpart], outputs=output)
    model.compile(
        loss='binary_crossentropy',
        optimizer='adam',
        metrics=['accuracy', auc_roc])
    return model
```

```
model_auc=get_model1(
    embedding_matrix,
    sentences_length,
    dropout_rate,
    recurrent_units,
    dense_size)

model_auc.summary()
```

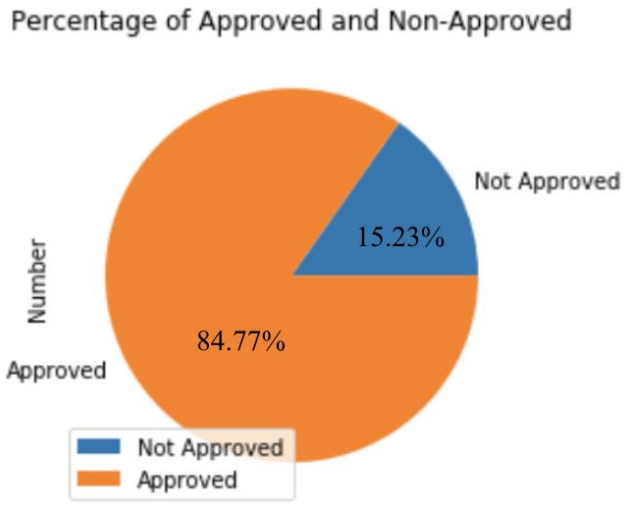
```
__________________________________________________________________________________________________
Layer (type)                    Output Shape         Param #     Connected to
==================================================================================================
text (InputLayer)               (None, 320)          0
__________________________________________________________________________________________________
embedding_5 (Embedding)         (None, 320, 300)     23672400    text[0][0]
__________________________________________________________________________________________________
spatial_dropout1d_5 (SpatialDro (None, 320, 300)     0           embedding_5[0][0]
__________________________________________________________________________________________________
bidirectional_5 (Bidirectional) (None, 320, 256)     329472      spatial_dropout1d_5[0][0]
__________________________________________________________________________________________________
nontext (InputLayer)            (None, 126)          0
__________________________________________________________________________________________________
capsule_5 (Capsule)             (None, 10, 16)       40960       bidirectional_5[0][0]
__________________________________________________________________________________________________
dense_13 (Dense)                (None, 64)           8128        nontext[0][0]
__________________________________________________________________________________________________
flatten_5 (Flatten)             (None, 160)          0           capsule_5[0][0]
__________________________________________________________________________________________________
dense_14 (Dense)                (None, 128)          8320        dense_13[0][0]
__________________________________________________________________________________________________
dropout_5 (Dropout)             (None, 160)          0           flatten_5[0][0]
__________________________________________________________________________________________________
concatenate_5 (Concatenate)     (None, 288)          0           dense_14[0][0]
                                                                 dropout_5[0][0]
__________________________________________________________________________________________________
dense_15 (Dense)                (None, 1)            289         concatenate_5[0][0]
==================================================================================================
Total params: 24,059,569
Trainable params: 387,169
Non-trainable params: 23,672,400
__________________________________________________________________________________________________
```

To get the best model, we adjust several hyper-parameters, including epoch number and class weight. I tried the epoch numbers 6, 8, 10 and 20. Six epochs gives the best performance and a large epoch number, such as 20, indicates a high risk of overfitting.

	6 epochs	8 epochs	10 epochs	20 epochs
Validation AUC	0.7703	0.7624	0.7698	0.7945
Test AUC	0.7816	0.7808	0.7760	0.7670

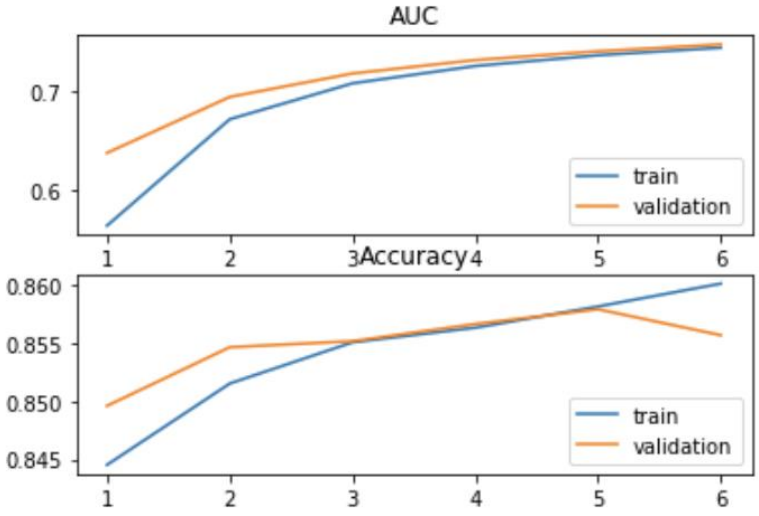
(The table shows the validation and test AUC for each epoch number.)

Moreover, regarding the imbalance of the data class, we add class weights when fitting the model to assign different weight for each class. We generate a class weight of [3.2826134, 0.5898436] for non-approval and approval class, which gives the Not-Approved class more weight to solve the problem of imbalanced data.



Note: This pie chart indicates that the “Approved” class takes up 84.77% while the “not-Approved” class only takes up. With such an imbalanced data, it is more likely to assign an essay to the “Approved” class.

After adding the class weights into the model, we see a slight improvement in the AUC. In the case of 6 epochs, the AUC increases from 0.78047 to 0.78155 after adding the class weights. Therefore, six epochs along with class weights so far give the best performance for the Bidirectional Recurrent Neural Network.



(The AUC and accuracy curves for validation and test data for the model with six epochs and class weights)

7.3 Performance Evaluation

The main metric to judge the performance of our models are Area Under the ROC Curve (AUC), which examines the performance of a classifier without fixing the threshold.

	Light GBM		XGBoost	Bidirectional RNN
Parameters Used	max_depth: 12 learning_rate: 0.025 num_leaves: 28 boosting_type: gbdt bagging_fraction: 0.85, bagging_freq: 5, verbose: 0, num_threads: 1, lambda_l2: 1.0, min_gain_to_split: 0		eta: 0.15, max_dep: 7, subsample: 0.80, colsample_bytree:0.80, seed: seed	batch_size: 256 recurrent_units: 64 dropout_rate: 0.3 dense_size: 32 sentences_length: 320 fold_count: 5 Epoch: 6 class_weights: [3.2826134, 0.5898436]
Text Model	Word2Vec	TFIDF	Word2Vec	Word2Vec
Training AUC Score	0.830955	0.92343	.9406	0.7437
Validation AUC Score	0.757288	0.782815	.79245	0.7703
Test AUC Score	0.72	0.7939	0.6646	0.7816
Training Time	256 secs	372 secs	487 secs	3605 secs
Predicting Time	21 secs	32 secs	55 secs	159 secs

In terms of AUC for the test data, the LightGBM with TFIDF and Bidirectional Neural Network gives the best prediction with AUC of 0.79 and 0.78 respectively. However, if we look at the training and predicting time, LightGBM would perform much better than Bidirectional Neural Network. Moreover, LightGBM can also show the prediction power for each attribute after training the model, which could give the DornorChoose.org more insights on the model and how to improve the model. While Neural Network is black-box algorithm which could not explain a lot of things happened inside compared to tree-based models. Therefore, we would recommend LightGBM with TFIDF as the best model for predicting whether the project will be approved or not.

8 Business Impact

The final solution will solve the two main problems DonorsChoose is facing:

- How to scale current manual processes and resources to screen 500,000 projects so that they can be posted as quickly and as efficiently as possible
- How to increase the consistency of project vetting across different volunteers to improve the experience for teachers

First, the number of applications they receive is increasing every year and the current screening process is manually vetting the applications by a team of volunteers. We know that Donors currently has 500,000 projects needed to be reviewed. Each project has four essays and we combine them into one complete essay. Each complete essay has 300 words on average and in total it is 1.5 million words. We assume a volunteer needs to spend at least 2 mins to read one complete essay and each volunteer usually work 6 hours a day. Our best model gives AUC of 0.79 calculated by Kaggle. The higher the AUC, the better the model is at predicting 0s as 0s and 1s as 1s. In other words, it has 79% capability to distinguish between approved proposals and rejected proposals. In terms of time, it could help DonorsChoose save 793,000 minutes and 2,202 volunteers.

Whether a proposal will be approved or not is based on our model instead of based on human judgment. Implementing our model could reduce subjectivity and bias by humans. Teachers could immediately get the result and they don't need to wait for a long time for their proposal to get approved. This increased their user experience.Besides, our feature selection could help them pick the the most crucial information to choose the right project for their contribution.

9 What to Improve?

- Hybridize RNN with other inherently explainable model to get better performance
- Explore other ways of converting text into vector
- Check for concurrency of the texts or required resources to special events that have occurred at a time
- Fluency and articulation of the texts can be an important factor if we could somehow measure it

One of the biggest problems we face is comparing each model's advantages and disadvantages to pick the best model. There are different approaches to building ML models for various text-based applications depending on what is the problem space and data available. Classical tree-based ML approaches like 'XGBoost' or 'LightGBM' can be implemented quickly and easily with higher accuracy. Deep learning techniques are giving better results for NLP problems like sentiment analysis and language translation. However, In the real word problems, we need to pay attention to the constraints of complex models like the bidirectional recurrent neural network we implemented. RNN is a black-box algorithm that could not explain lots of things compared to other tree-based models. Ideally, we can avoid the black-box problem from the beginning by

developing a model that is explainable by design. It might be possible to hybridize an inherently explainable modeling approach with a complex black-box method.

The know that word embedding can be obtained by training a lot of language models. We used Word2Vector and TF-IDF in our model. Compared to TF-IDF, Word2Vector could capture the context of a word in a document, semantic and syntactic similarity and relation with other words, but it turns out to give us a lower AUC than TF-IDF. Therefore, it might be a good choice if we can explore other popular language model such as GloVe, fasttext, etc. Another method is to train our own context specific world embedding model instead of the pre-trained ones.

It might be useful to check the concurrency of the texts or required resources to special events that have occurred at a time. Maybe because of some events, some proposals are being accepted more easily, due to public awareness or the hotness of some topics. Besides, fluency and articulation of the texts are important factors in text mining. They might increase our prediction accuracy if we could somehow measure them.

10 Next Step

- Build a reliable pipeline to streamline the screening process with big data tools
- Provide recommendations on how to improve proposal quality

From a technical standpoint, we can build a complete pipeline from data preparation to prediction by leveraging big data tools such as spark machine learning which could give real time responses and increase operational efficiency.

From a business viewpoint, after we know what projects will be rejected. We can do more to deal with rejection. By giving recommendations and feedback to teachers on how to improve the proposal to get a pass, the proposal quality and conversion chances will be further enhanced.

11 Appendix (Kaggle Submission Score)

Bidirectional RNN

fiddle_6epoch(cw).csv 2 days ago by 115020155 add submission details	0.77103	0.78155

LightGBM with TFIDF

Submission and Description	Private Score	Public Score
submission.csv 18 hours ago by Elaine add submission details	0.78233	0.79388

LightGBM with Word2Vec

submission.csv a few seconds ago by 115020155 add submission details	0.72159	0.72336

XGboost

xgb_1-3.csv 21 hours ago by Maya Carnie xgb again #3	0.66240	0.66456