Predicting the Success or Failure of Kickstarter Campaigns

Joseph Liu, Keenan Park, and Chris Yi

Computer and Information Science, University of Pennsylvania, Philadelphia, PA, USA
Email: {joeliu, keenanp, tianzeyi}@seas.upenn.edu

Abstract

Fueled by media coverage of massive success stories, Kickstarter has become a popular crowdfunding platform where creators can seek funds to turn their ideas into products. However, only a third of projects manage to reach their fundraising goals and little is known about the ingredients required to make a successful Kickstarer campaign. To solve this problem, we analyzed over 380,000 Kickstarter campaigns from 2009 to 2018 and ran a plethora of machine learning algorithms to predict whether a campaign will be a success or not. In general, accuracy was low across all models due to underfitting. Tree-based models performed better with XGBoost as the winner with an accuracy of 70.23% and recall of 43.64%. Our results indicate a need for more features by way of feature engineering or scraping more data to improve the performance of our models.

I. Motivation

Online crowdfunding has empowered creators and entrepreneurs to raise money on their own terms. In 2016, Elena Favilli and Francesca Cavallo launched a Kickstarter campaign to fund the production of their new children's book Good Night Stories for Rebel Girls. They set out to raise a modest $40,000 to print 1000 copies, but ultimately received a record-breaking $675,614. The book has since sold over one million copies, and the project has evolved into a burgeoning media company, spanning over 70 countries.

Success stories like that of Favilli and Cavallo have inspired thousands of others to launch their own Kickstarter campaigns, in hopes of attaining similar results. In total, over 4.6 billion dollars have been pledged toward Kickstarter projects, but only 37% of them manage to reach their fundraising goals. We do not believe that most campaigns fail by mere chance, so we intend on using machine learning to reveal the underlying causes.

We were fortunate to obtain a large dataset detailing over 380,000 Kickstarter campaigns from 2009 to 2018. To uncover latent correlations in the data and predict the success or failure of future campaigns, we will use a host of classification techniques ranging from vanilla logistic regression to modern boosting algorithms such as XGBoost. In the end, we hope that the success predictors we identify will be useful for prospective entrepreneurs and Kickstarter users alike.

II. Related Work

In this section, we will outline prior attempts at predicting Kickstarter campaign performance. Tran et al. [1] collected data from Kickstarter and Twitter to identify project success predictors and determine the expected amount of money an arbitrary project would raise. They used 5-fold cross-validation to evaluate a naïve Bayes classifier, a random forest, and an AdaboostM1 classifier. Of the three, AdaboostM1 was the best performer, achieving an accuracy of 76.4% and an AUC of 0.838. Their models indicated that successful projects have shorter project durations, modest funding goals, and active social media engagement. The location of the founders, however, appeared to be uncorrelated with the success of the project.

Lewis [2] experimented with a different set of models to achieve a similar goal. She used an F1 score to evaluate vanilla logistic regression, logistic regression with PCA and parameter optimization, a random forest classifier, and XGBoost. All of these models were able to achieve an accuracy of roughly 70% and were better at predicting successes than failures. Since the random forest and XGBoost classifiers appeared to overfit, logistic regression was deemed the best model.

Ultimately, Lewis found that successful projects often had modest goals, the “staff pick” designation, a connection to dance or comics, and/or founders from Hong Kong. Conversely, failed projects often had ambitious fundraising goals, a connection to food or journalism, and/or founders from Italy.

III. Dataset

We obtained our dataset from Kaggle. It is maintained by Mickael Mouille, and it contains 378,661 campaigns dating from May 3, 2009 to March 3, 2018. In total, there are 15 features: ID, name, category, main_category, currency, deadline, goal, launched, pledged, state, backers, country, usd_pledged, usd_pledged_real, and usd_goal_real.

A. Differences between pledged quantities

Most campaigns were pledged in US dollars, but not all. The pledged column simply denominates the quantity of whatever currency that campaign was created in, whilst the usd_pledged column is the conversion to US dollars done by Kickstarter. The maintainer of this dataset also added an additional column, usd_pledged_real, in which the conversion is done by Fixer.io API instead.

What we have found by looking at the data is that the pledged column does not seem to be reliable at all. To illustrate this, the 3rd campaign in the picture below shows the pledged amount as £94,175, whilst the usd_pledged is only $57,763.78! This clearly doesn't make sense because the dollar amount should be more than the pound amount. However, the usd_pledged_real amount of $121,857.33 seems to be much closer to what we would expect.

There are several other instances where the usd_pledged column does not seem to accurately reflect the actual pledged amount in US dollars. Because of this, we have chosen to drop pledged and usd_pledged and use the more accurate usd_pledged_real column instead.

B. Adding duration, launched day, and deadline columns

The data has *launched* and *deadline* columns, which refer to the date and time the campaign was launched and ended. Even though it could be gleaned implicitly, we felt it was important to add *duration* as a standalone column, which is simply the difference between *deadline* and *launched*.

In addition to this, we converted *launched* and *deadline* to *launched_dayofyear* and *deadline_dayofyear* for two reasons: 1) it's easier to compute a column that's an integer instead of a DateTime variable and 2) our goal is to predict future campaigns and we can't create campaigns in the past.

C. Dealing with missing and/or nonsensical values

Using Panda's info() function, we are able to quickly find out if our dataset contained any null values [GRAPHIC]. The important thing to notice in this table is that name and *usd_pledged* contain some null values. Thankfully, we can drop the *name* column as we do not suspect that name will be an important feature in our models. We have also explained in the previous section that *usd_pledged* would be dropped in favor of *usd_pledged_real.*

Non-null values can be erroneous as well. For example, some campaigns had mistakenly recorded their launch date as one prior to 2009, the year in which Kickstarter was founded. In particular seven campaigns had a launched date of 1970-01-01 01:00:00, 3979 campaigns had a puzzling country value of 'N,0"', and a number of campaigns had a state other than *successful* or *failed*.

The launch date errors may be the result of an integer overflow, since 1970-01-01 is the start of the Unix epoch 1. In any case, because only seven campaigns had this mistake, we excluded all seven from the final dataset.

Fixing country values was more of a challenge. Roughly 1% of our entire dataset consisted of campaigns with nonsensical country values. With a bit of detective work, we were able to figure out which country the campaign came from by looking at the currency used. There was one caveat, however: the euro, which is used by several countries. To deal with this, we decided to use the most frequently occuring euro-using country, which was Germany.

In the next section, we will describe how we dealt with the last scenario, in which the state column contained labels other than *successful* and *failed.*

D. Campaign State

Ultimately, we would like to predict whether a campaign was a success or not. The *state* column gives us the final result of the campaign. As it turns out, there are quite a few states the campaign can end up in. No description was given to explain their meanings, so we could only make assumptions and infer:

- *Failed:* We assume the campaign was not able to reach its goal within the alotted time.
- *Cancelled:* We assume the campaign was cancelled by the creators for unknown reasons.
- *Live*: We assume the campaign was suspended by Kickstarter for unknown reasons.
- *Undefined*: Unable to infer any meaning.

State	Num. of Campaigns	% of Campaigns
Successful	133,956	35.38
Failed	197,719	52.22
Cancelled	38,773	10.24
Undefined	3,562	0.94
Live	2,799	0.73
Suspended	1,845	0.49
Total	**378,654**	**100**

The table above shows a breakdown of all the campaign states. The first thing to note is that only about one-third of campaigns actually succeed. More than half of them fail. The rest have not or do not reach their goal for other reasons. Because the labels *failed, cancelled,* and *suspended* all relate to failure in some way, we decided to re-lable these as *failed.*

The challenge now is how to deal with the *live* and *undefined* labels.

We exclude the *live* data for two reasons: (1) we could not not easily assimilate it into the rest of our data and (2) it only amounted to 0.73% of our dataset.

For the data labeled *undefined*, we reimputed the campaign state as *successful* if the pledged amount met the goal amount and *failed* otherwise. Out of 3,562 *undefined* rows, 1,908 rows were reimputed as *successful*, and 1,654 were reimputed as *failed*.

Lastly, we defined a successful campaign as one that was able to reach its goal amount. It turned out that 55 campaigns reached their fundraising goal but were not labeled *successful*.

The question was whether to include this data or not. Technically it counts as a success as we have defined, but for some unforeseen reason, the campaign creators must have cancelled it. Perhaps they realised they couldn't carry through what they intended to make and didn't want to defraud their investors? Or maybe they got cancelled by Kickstarter instead for promising something they could not deliver? Because of the possibility that the campaign creators may have gamed their campaign, in addition to the fact that 55 campaigns only represented a small sliver of our data, we decided to exclude these rows.

E. Dropping additional columns

As we explained previously, we dropped the *name* and *usd_pledged* columns. We also omitted the following:

- *ID*: We don't expect the campaign ID to correlate with the campaign state.
- *goal*: This is less reliable than *usd_goal_real.*
- *usd_pledged_real:* Machine learning algorithms will be able to implicitly learn the campaign state when the pledged amount exceeds the goal amount.
- *backers*: We left this out for two reasons: (1) it can implicitly give away the campaign state, and (2) the number of backers is not something the campaign creator can control, and is therefore not useful to founders.

All of this preprocessing reduces our dataset to 373,875 rows and 9 columns.

F. One-hot encoding of discrete features

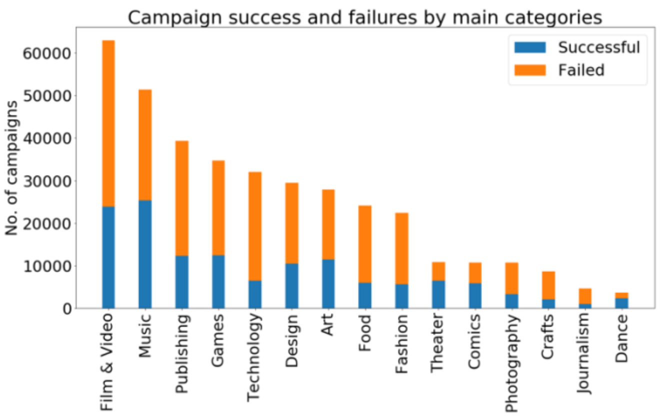
Most of our features are discrete, and given none of them imply and kind of relationship (like day of the week), it made sense to convert them to one-hot encoding. Because of the large amount of categories and main categories, this converted the final dataset to 373,875 rows and 206 columns.

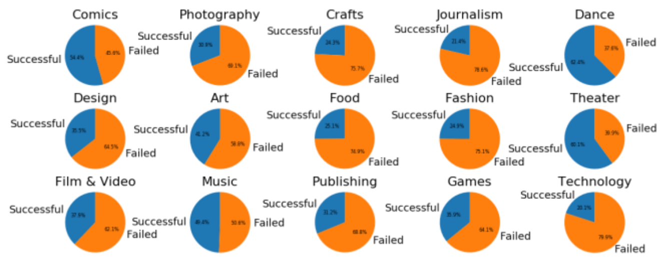
G. Analysis of the dataset

Now that we have dealt with missing and erroneous data, it is time to analyze it and see what insights can be made. After reconfiguring our state column to just two labels, *successful* and *failed*, the final breakdown is:

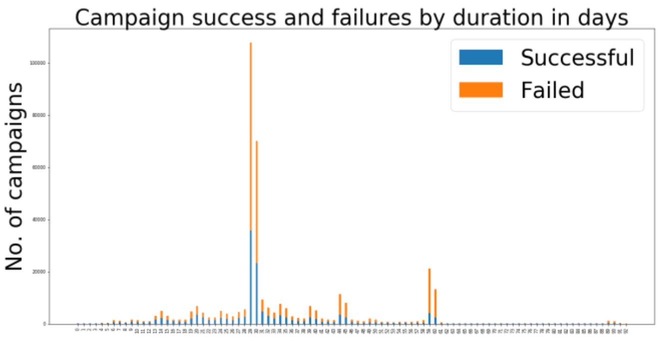
State	**# of Campaigns**	**% of Dataset**
Succesful	135,610	36.1
Failed	240,190	63.9
Total	**375,800**	**100**

We also examined the main categories, and found that there are 15 of them. Nearly all, with the exception of technology, are related to entertainment or media. The most popular ones are Film & Video, Music, and Publishing, while the least popular ones are Dance, Journalism, and Crafts.

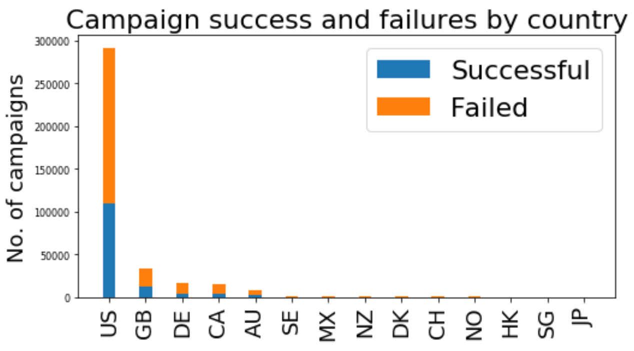




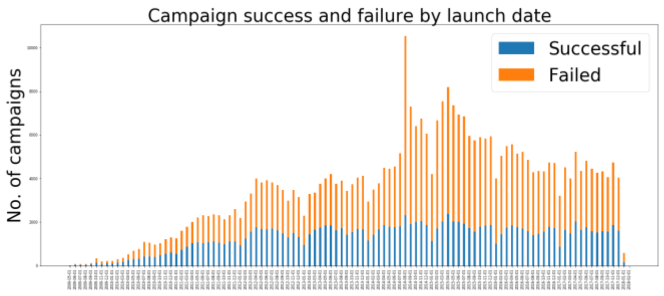
From the pie charts, we can see that the categories with the highest rates of failure are Journalism, Technology and Crafts. On the other hand, Theatre, Dance and Comics have the highest rates of success. This brings some interesting insight considering the fact that those 3 categories rank are among the least popular categories.



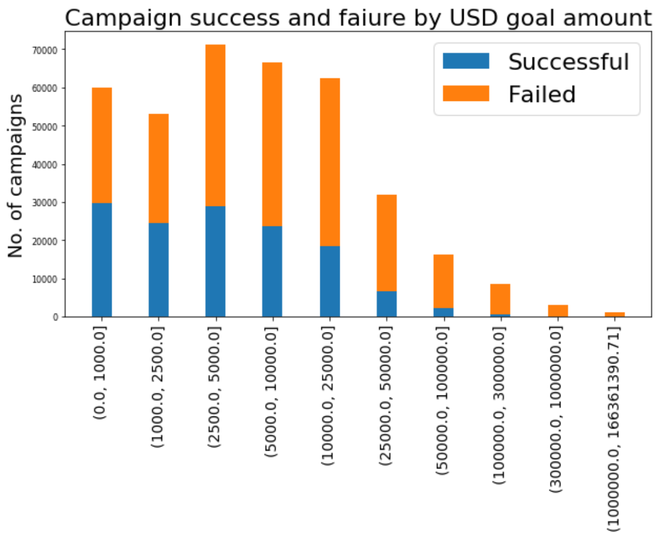
The bar chart above gives us the distribution by duration in days. By far, the majority of campaigns choose a duration of 29 or 30 days. This is likely due to the fact that the recommended duration from Kickstarter is 30 days. By country, the distribution follows a similar pattern where most of the campaigns are concentrated in one point: the U.S. It seems like the U.S. also has the highest rate of success as well.



The campaigns range from May 2009 to February 2018. The total number of campaigns peaked between August 2014 to April 2015.



Despite this, it’s interesting to note that the number of success has remained relatively flat at around 2000 successes per month. A plausible explanation for this could be that the number of backers on Kickstarter have remained more or less the same, whilst the number of creators saw a massive surge around 2014/15.



The goal amounts ranged from $0 to $166,361,390. Most campaigns had a goal somewhere between $2,500 to $10,000. The graph also shows that as the goal amounts increased, the rate of success decreased, which is in line with what we would assume.

H. How noisy is our data?

Following this blog post [4], we used the library *featexp* to analyze the correlations of features between training and test data. For example, usd_goal_real showed that as the goal amount increase, the average state value decreased, which is what we expect. But additionally, it allows us to show if a feature holds the same trend over the test data as it does over the training data. If it doesn’t, then it displays a lower correlation and leads to overfitting.

Our features generally show a high correlation above 99%. The two lowest features were launched_dayofyear and deadline_dayofyear, which showed correlations above 91%. If they were lower, we could decide to drop these features. But because 91% is still considered pretty high, and we don’t have that many features to begin with, we decided to keep these two features in.

IV. Problem Formulation

We want to use machine learning to predict whether or not a Kickstarter campaign will meet its fundraising goal. Our ideal outcome is to identify a set of success predictors that founders can use to assess the viability of their project. A success metric is the number of successful Kickstarter projects properly predicted by our model, and success means predicting roughly 80% of the successful campaigns in the test set, which would surpass the current state of the art.

Our problem is best framed as a supervised, binary classification problem. As such, we will use logistic regression as our baseline method. If our accuracy goals are not met, we will experiment with decision trees, random forests, Adaboost, XGBoost, and neural networks.

V. Methods

A. Data Preparation

As part of the preprocessing step, we have divided the dataset into three groups: training (80%), validation (10%), and test (10%). The test set will be set aside after exploratory data analysis. The training set will be used to generate models and perform cross validations for hyperparameters within each model. Next, the validation set will be used to select the best model for ensemble of ensembles. Finally, the output models will be evaluated on the test set.

In this project, we will build and optimize five models: 1) a logistic classifier 2) a boosting model, selected from Adaboost and Gradient Boosting; 3) a decision tree, compared with a random forest made of stumps; 4) an ensemble of ensembles; and 5) a deep neural network.

B. Logistic Regression: Baseline

To start, we have chosen Logistic Regression as our baseline, because Logistic Regression is the most basic model and suitable for most binary classification problems. Moreover, it can serve as an indicator of whether our data are linearly separable, affecting classification strategies later on (see below). To perform a Logistic Regression, we intend to call on sklearn.linear_model.LogisticRegression in the scikit-learn package, using L2 penalty and other default parameters. We will check and, if necessary, increase max_iter to ensure convergence.

C. Logistic Regression Optimization

Once a baseline has been established, we will improve upon our logistic classifier model by tweaking the hyperparameters. To start, we will perform a grid search over different feature transformations, such as MinMax and log-scaling, so as to make features Gaussian. We will also apply qualitative knowledge about the various features and standardize them. Furthermore, we will add a regularization term, starting with L2: since we only have 10 features, feature selection is less of a concern than one feature taking up too much weight. Therefore, L2 penalty, which shrinks large weights the most but preserves all features, is the most appropriate regularization method. Nevertheless, we will still try other penalty terms in case they produce better results. We will then perform a 5-fold cross validation using the following penalty coefficient: {0.1, 1, 10, 100, 500, 1000}. Both coefficient and penalty term selections will be done via grid search, a built-in functionality on Scikit Learn. Once finished, we will have a new Logistic Regression model.

D. XGBoost Classifier and Optimization

We thought about the relationship between n and p; since n is relatively large compared to p, we decided to perform XGBoost, a state-of-the-art non-deep learning method for medium-sized datasets.

E. Adaboost Classifier and Optimization

Next, we will build two more ensemble models: Adaboost and Random Forest. We will also attempt Gradient Tree Boosting, although it will likely be too slow for large datasets such as this one. For Adaboost, we will use a tree stump as a base learner - it is most appropriate for classification and weak enough to ensure robust ensemble results. Next, we will use sklearn.ensemble.AdaBoostClassifier using the default base learner, i.e. DecisionTree- Classifier (max_depth=1), as well as other hyperparameters (n_estimators=50 and learning_rate=1.0. However, it is possible that the model will either converge before 50 iterations, resulting in overfit, or not yet converge at 50 iterations, resulting in weak predictions. To address this concern, we will perform cross validation: set learning_rate constant and use a range of n_estimators, selecting the n_estimators that leads to the lowest cross validation error. Further improvements will be made by trying out various base_learners, such as SVC. The best combination of methods and hyperparameters will be used as the final Adaboost model.

F. Random Forest Classifier and Optimization

As for a Random Forest, we will use sklearn.ensemble.RandomForestClassifier. Just like before, we will start with the default values, i.e. max_depth = None, max_features = auto, min_samples_leaf = 1, min_samples_split = 2, and n_estimators = 100. Then, we will vary these parameters and use cross validation to select the best combination. Importantly, we will compare the results between Random Forest and Adaboost.

G. Deep Neural Network, Optimization, and Comparisons

Finally, we will build a deep neural network using SKLearn’s MLP Package. We have good reasons to try this method: our dataset is large enough, the “black-box” aspect is of little concern to us, and, unlike previous methods, neural networks do not assume function form and eliminate the need for data transformation. For our purposes, we will build a supervised deep neural net, with ReLU as the activation function and log-likelihood as the loss function. We will begin with a simple model that has only one fully-connected hidden layer. In case the model is too slow, we will speed it up with mini-batch gradient descent.

VI. Experiments and Results

For all our models, we performed a grid search with 5-fold cross validation to select the optimal hyperparameters. In an ideal world, we’d simply put all these parameters in a single dictionary and find the optimal combination of parameters by running GridSearchCV. However, doing so is extremely computationally expensive. Consequently, we opted for two alternatives: (1) GridSearchCV using 5-fold cross-validation one parameter at a time and (2) randomized search.

In the former, we find the optimal value for a parameter, fix that parameter to the optimal value, and iterate grid search on the next parameter. This approach is a greedy one and not guaranteed to find the optimal *combination* of parameters, but the tradeoff meant that were able to train our models more efficiently.

In the latter, namely randomized search, we define a distribution for each parameter, and the RandomizedSearchCV function performs 5-fold cross validation on a set of values selected at random from the distributions. This was especially useful for tuning the hyperparameters of XGBClassifier.

Once we found the best parameters for each model, we then evaluated them using the following metrics:

- **Accuracy**: The model is used to make predictions on the test set. The predicted values are compared against true test labels *(successful* or *failed*). This is the most direct, easy-to-interpret indicator for the performance of classification problems.

- **F1**: 36% of all campaigns are successes and the rest are failures. Due to this class imbalance, we would use the F1 to evaluate the performance, which is a balance between precision and recall.
- **AUC-ROC**: AUC tells us how good the model is capable of distinguishing between classes.
- **Confusion matrix**: This is the summary table for TP, TN, FP, and FN.

A. Logistic Regression

For logistic regression, we performed a greedy grid search with 5-fold cross validation to tune two hyperparameters: *C* and the penalty. In sci-kit learn's implementation of logistic regression, *C* is the inverse of of the regularization coefficient typically denoted by lambda. For the penalty, the options were L1 and L2.

To start, we implemented logistic regression with only default parameter values. This resulted in an accuracy score of 63.84%. Then, we searched over both penalties, and a set of 20 *C* values from 0 to 2.

In this iteration of grid search, we found that L1 performed better than L2, and 0.7368 was the optimal *C* value. We performed grid search once more, holding L1 constant and narrowing the search space for *C*.

Ultimately, the best set of parameters were the following:

1. C = 0.6631578947368421
2. penalty = l1

These values generated an accuracy score of 68.67%, an increase of roughly five percent.

B. XGBoost Classifier

As we alluded to previously, we employed a randomized search to tune the hyperparamters for the XGBoost classifier, but we began with a vanilla implementation like we did with logistic regression.

Interestingly, the default hyperparameter values for XGClassifier (the sci-kit wrapper for xgboost) performed quite well. In particular, it achieved an accuracy of 69.08% on a 5-fold cross validation on the training set. This led us to believe that the optimal hyperparameter values lied somewhere in the neighborhood of the default values. As a result, we selected distributions for each parameter value such that they did not stray too far from the default values.

We then performed 500 search iterations over these distributions. In the end, we obtained a model that achieved a 70.23% accuracy on the test set, a figure that would prove to be the best among all models tested. The parameter values for the best model were the following:

1. max_depth = 9
2. learning_rate = 0.04635328483253784
3. n_estimators = 482
4. min_child_weight = 2
5. colsample_by_tree = 0.5126256032832679
6. subsample = 0.5404867551864891

C. Adaboost Classifier

For Adaboost, we performed a greedy grid search with 5-fold cross validation to tune two hyperparameters: learning_rate and base_estimator.

First, we felt a decision tree stump to be the most optimal estimator, since it's weak enough for AdaBoost. However, what remains to be decided is the size of the stump. With this question in mind, we did a search over various max_depth of the tree, from 1 to 5. We found that while max_depth = 5 gives slightly better mean CV accuracy than max_depth = 1, the difference is negligible (0.695 vs.0.689). For this reason, we decided to use max_depth = 1, in line with the thinking that tree weak stumps produce a strong ensemble. Next, we did a search over learning_rate from [0.1, 0.25, 0.5, 0.75, 1] and found 0.75 to be the most optimal.

Overall, the performance of AdaBoost for the test set is 0.693 (the same model produced 0.691 accuracy on the training set). The following is the confusion matrix:

	Predicted Failure	**Predicted Success**
Actual Failure	20,545	3,406
Actual Success	8,060	5,373

With that in mind, the optimal parameters were: (1) base_estimator = DecisionTreeClassifier(max_depth = 1), and (2) learning_rate = 0.75

D. Decision Tree

We used sklearn's DecisionTreeClassifier and trained on these parameters in the following order:

1. min_samples_split = [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05]
2. min_samples_leaf = [0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005]
3. max_depth = [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
4. max_features = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]

The above order was chosen because an empirical study found min_samples_split and min_samples_leaf to be the most responsible for the performance of the final trees [5].

With that in mind, the optimal parameters were:

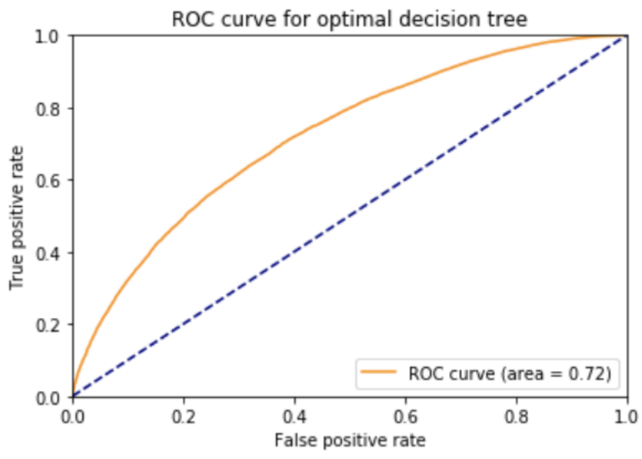
1. min_samples_split = 0.005
2. min_samples_leaf = 0.0001
3. max_depth = 18
4. max_features = 0.9

	Default Dec. Tree	**Opt. Dec. Tree**
Accuracy Score	61.49	69.37

By optimizing the decision tree, we were able to improve accuracy by 7.88%. We also found that out of all 206 features, the most important feature was *usd_goal_real*. Using the optimal decision tree gave the following confusion matrix:

	Predicted Failure	**Predicted Success**
Actual Failure	20,479	3,471
Actual Success	7,981	5,457

We also obtain the following ROC curve:



E. Random Forest

We used sklearn's RandomForestClassifer and trained on these parameters using the greedy algorithm:

1. min_samples_split = [0.0001, 0.0005, 0.001, 0.005, 0.01, 0.05]
2. min_samples_leaf = [0.00001, 0.00005, 0.0001, 0.0005, 0.001, 0.005]
3. max_depth = [3, 6, 9, 12, 15, 18, 21, 24, 27, 30]
4. max_features = [0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
5. n_estimators = [200, 400, 600, 800, 1000]

The optimal parameters we found were:

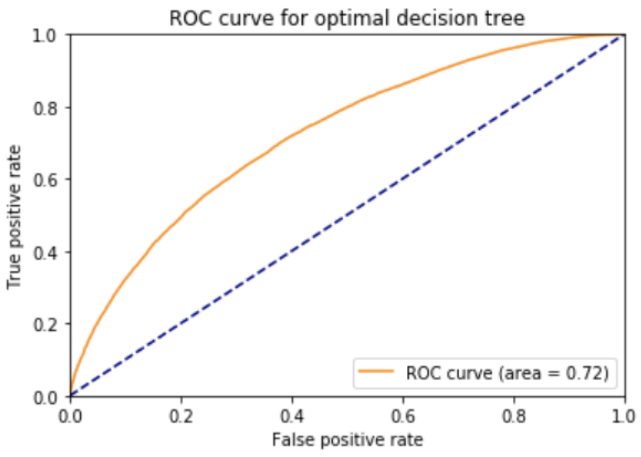
1. min_samples_split = 0.0005
2. min_samples_leaf = 0.00001
3. max_depth = 1000
4. max_features = 0.2
5. n_estimators = 1000

	Default Rand. Forest	**Opt. Rand. Forest**
Accuracy Score	66.41	70.21

By optimizing the random forest classifier, we were able to improve accuracy by 3.8%. Like the decision tree previously, we found the most important feature was *usd_goal_real*. Using the optimal random forest classifier gave the following confusion matrix:

	Predicted Failure	**Predicted Success**
Actual Failure	20,586	3,364
Actual Success	7,774	5,664

And, as we did with the decision tree, we obtain the following ROC curve:



E. Neural Network

For Neural Network, we performed a grid search with 5-fold cross validation to tune two hyperparameters: activation and alpha. For activation, i.e. the activation function used by the hidden layers, we tested the following: ['identity', 'logistic', 'tanh', 'relu'].

Similarly, we did a search over alpha, the L2 penalty, from [0.0001, 0.001, 0.01, 0.1, 0.25, 0.5, 0.75, 1] and found 0.001 to be the most optimal. The performance of MLP over the test set is 0.680 (the same model produced 0.679 accuracy on the training set). The following is the confusion matrix:

	Predicted Failure	**Predicted Success**
Actual Failure	21,119	2,751
Actual Success	9,210	4,228

VII. Conclusion and Discussion

In general, test accuracy was not exceedingly high and remained in the high 60s for all the models we tested. The training scores for each model also remained in the same region between 65-70%, so it seems like the main issue is that our models are underfitting.

Despite the fact that our final dataset had 206 features, many of these features were one-hot encodings of categorical features, so really, we only had 8 distinct features.

	Accuracy (%)	Precision (%)	Recall (%)	AUC (%)	F1 (%)
Logistic Regression	69.09	60.97	38.91	72.00	47.51
Decision Tree	69.37	61.23	40.61	72.00	48.80
Random Forest	70.21	62.73	42.15	74.00	50.42
AdaBoost	69.50	61.23	40.02	72.00	48.40
XGBoost	70.23	62.24	43.64	74.00	51.30
MLP	68.00	60.58	31.46	57.00	41.41

A. Model Comparison

The top winner out of all the models we tried was XGBoost, followed by random forest and AdaBoost. It seems like tree-based methods were superior here, which probably implies that our data is non-linear. To confirm that hypothesis, we ran logistic regression on the single most important feature, usd_goal_real, and only got 64% test accuracy.

Aside from accuracy, XGBoost also performed best on recall, AUC (tied with random forest) and F1. The only metric it didn't win was precision, which was topped by random forest.

This brings us to the next point of our discussion: which metric do we care about most here? Firstly, we've already seen that our data is generally unbalanced between positives and negatives, with only a third of campaigns coming out as successes. Therefore, an F1 score would be a better metric over AUC as AUC averages over every threshold.

Secondly, which of the two do we care more about: false positives or false negatives? A false positive is when we predict a campaign to succeed yet it fails, whilst a false negative is when we predict a campaign to fail yet it succeeds.

In the spirit of entrepreneurship, we believe in the mindset of positivity, because after all, if a campaign creator fails, then they can always try again. But on the other hand, if they don't try at all, then they can never succeed. Or as Wayne Gretzky put it succinctly: "You miss 100 percent of the shots you don't take."

With that in mind, we believe optimizing for false negatives is more important in our case. No one wants to be the person who told someone they wouldn't succeed, only to be proven wrong later on. If we take recall as our primary metric, then XGBoost remains the clear winner.

B. Lessons Learned

For all of us, this was our first end-to-end data science project. The biggest takeaway we learned is that no amount of tweaking of models can improve the accuracy by a significant amount if the dataset is too simple, as it was in our case.

In fact, we were surprised to learn that most of our time was spent exploring the data and making simple analyses into the features rather than actually using machine learning techniques we had learned in class.

The problem is that our dataset contains too little distinct features. Adding features by way of feature engineering seems like an essential tactic employed by many to when it comes to winning data science competitions [6].

An alternative to feature engineering would be to simply get more data. A Kickstarter campaign contains much more information than what was contained in our dataset, for example, a vital one that was missing is the sales pitch itself.

C. Future Extensions

We've identified that the main problem to our performance was simply a lack of features, so an extension to our project would be to simply get more features. We've already mentioned feature engineering as one method. Another method would be to dig up the old campaigns using the Kickstarter IDs and scrape data additional features that was not included. These features include:

- Sales pitch or Story
- Information about Risks and Challenges
- Information about the creators - where are they from, how many campaigns they have created in the past (and whether those were successful or not)

Having blurbs on a project's Story and Risks and Challenges, would also allow us to run NLP algorithms to gain further insights. For example, many sales pitch specialists often advocate the use of "power words", so wouldn't it be nice to find out first hand which words these were?

VIII. References

1. Thanh, R., M., Chung, Lee, & Kyumin. (2016, July 22). How to Succeed in Crowdfunding: A Long-Term Study in Kickstarter. Retrieved from https://arxiv.org/abs/1607.06839

2. Lewis, L. (2019, April 09). Using machine learning to predict Kickstarter success. Retrieved from https://towardsdatascience.com/using-machine-learning-to-predict-kickstarter-success-e371ab56a743

3. Jain, A. (2019, September 13). Complete Guide to Parameter Tuning in XGBoost (with codes in Python). Retrieved from https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/

4. Pawar, A. (2019, April 24). My secret sauce to be in top 2% of a kaggle competition. Retrieved from

https://towardsdatascience.com/my-secret-sauce-to-be-in-top-2-of-a-kaggle-competition-57cff0677d3c

5. Mantovani, Gomes, R., Cerri, Ricardo, Junior, Barbon, S., . . . André Carlos Ponce de Leon Ferreira. (2019, February 12). An empirical study on hyperparameter tuning of decision trees. Retrieved from https://arxiv.org/abs/1812.02207

6. Fogg, A. (2019, December 09). Anthony Goldbloom gives you the secret to winning Kaggle competitions. Retrieved from https://www.import.io/post/how-to-win-a-kaggle-competition/