Machine Learning Engineer Nanodegree

Capstone Project

Robert Martinez
July 28th, 2018

I. Definition

Project Overview

Crowdfunding – namely, the practice of raising money for a venture from a large base of small donors via the internet - has become a popular form of funding for small, creative projects. Kickstarter was one of the earliest crowdfunding platforms to become a household name in this space, and as a result has a large dataset of projects with which we can explore the factors that contribute to the success or otherwise of fundraising efforts.

Gaining a better understanding of the success/failure factors for crowdfunding projects could benefit all sides of the market, through a reduction in the rate of failed (and cancelled) projects. Given a sufficiently accurate model, Kickstarter could predictively suggest adjustments in project specifications to project proposers (goal amount conditional on project category, or campaign length, for example).

Previous work in this domain includes papers such as Kamath and Kamat (2016)[1] as well as more informal analyses (see here).

The dataset to be used for this project comes was retrieved from Kickstarter by crowdfunding enthusiast Mickaël Mouillé, and reviewed by data science platform Kaggle. It can be obtained by clicking here. Two datasets are included, one featuring roughly 324,000 projects up to February 2016, and the other featuring **378,661** projects up to January 2018. The latter set was used for this capstone project.

The dataset gives information about projects along 15 dimensions, including:

- Project Name

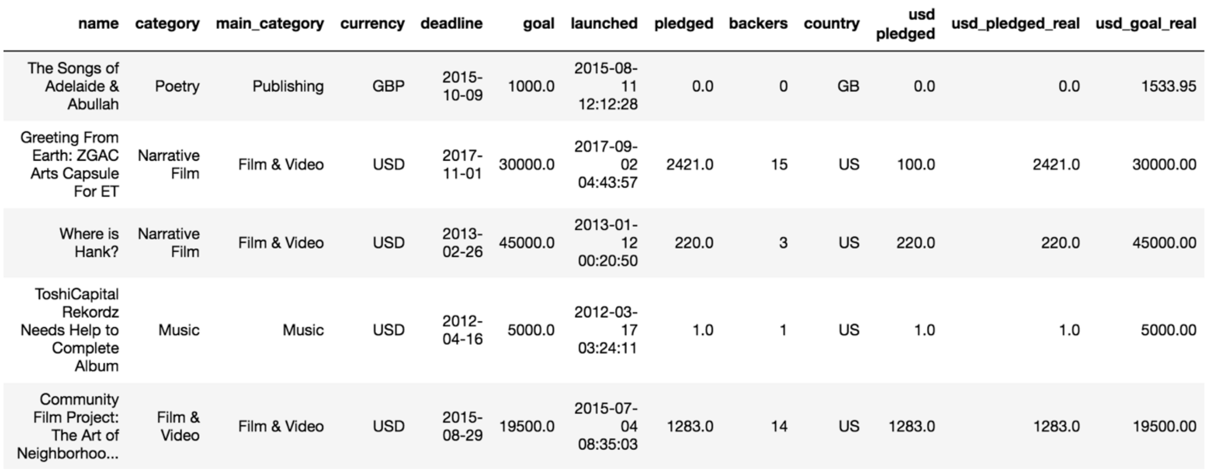
[1] Kamath, R.S. and Kamat, R.K. 2016. Supervised Learning Model for Kickstarter Campaigns With R Mining. International Journal of Information Technology, Modeling and Computing (IJITMC) Vol. 4, No.1.

- Project Category (Specific Category and Main Category)
- Project Launch Date and Deadline
- Country
- Currency
- Goal (in specified currency and USD if they differ)
- Amount Pledged (in specified currency and USD if they differ)
- Number of Backers
- Project State (Successful, Failed, Cancelled, Suspended, Undefined)

The target feature is **Project State**. The distribution of project states is somewhat unbalanced, i.e. around 35% of projects are classified as successful, while roughly 65% are classified as unsuccessful – either failed, cancelled, suspended, or undefined[2], with a small group (<0.1%) of projects still live at the time when the data was collected.

Problem Statement

This project aims to discover a classification model that can accurately predict the state of Kickstarter projects (success or failure) given the project characteristics. In other words, for a given project, given a set of input features/project characteristics *x*, I aim to train a classifier *f(x)* such that it accurately predicts the target variable *y*, which indicates whether the project was successfully funded or not.



name	category	main_category	currency	deadline	goal	launched	pledged	backers	country	usd pledged	usd_pledged_real	usd_goal_real
The Songs of Adelaide & Abullah	Poetry	Publishing	GBP	2015-10-09	1000.0	2015-08-11 12:12:28	0.0	0	GB	0.0	0.0	1533.95
Greeting From Earth: ZGAC Arts Capsule For ET	Narrative Film	Film & Video	USD	2017-11-01	30000.0	2017-09-02 04:43:57	2421.0	15	US	100.0	2421.0	30000.00
Where is Hank?	Narrative Film	Film & Video	USD	2013-02-26	45000.0	2013-01-12 00:20:50	220.0	3	US	220.0	220.0	45000.00
ToshiCapital Rekordz Needs Help to Complete Album	Music	Music	USD	2012-04-16	5000.0	2012-03-17 03:24:11	1.0	1	US	1.0	1.0	5000.00
Community Film Project: The Art of Neighborhoo...	Film & Video	Film & Video	USD	2015-08-29	19500.0	2015-07-04 08:35:03	1283.0	14	US	1283.0	1283.0	19500.00

Figure 1: Sample of Input Features

Given the variety of input features, which consists of string, categorical and numerical data (see **Figure 1**), a number of data preprocessing steps will be required to deal with missing values, redundant features, and conversion of data to numerical formats so they can be run through machine learning algorithms.

[2] The 'undefined' category will be treated with in the data preprocessing section.

Given the problem statement, I intend to train at least three models to try to discover an effective solution to the classification problem: a Support Vector Classifier (or SVM), Adaptive Boosting (AdaBoost) with a decision tree base learner, and XGBoost. The performance metrics of these models will be compared and analysed to determine the best model, and the best performer will be optimised using Grid Search cross validation, with F1 scoring to determine a final solution.

Metrics

Given that the task of predicting successful or unsuccessful Kickstarter projects is a binary classification task, an appropriate evaluation metric to be used is the **accuracy score**, i.e. the ratio of correctly predicted observations to total observations. This simple metric allows comparison to other models, namely the benchmark model specified above, and is easy to interpret.

However, in the model selection process, I will also make use of the **F1 score** to judge the appropriateness of different models and hyperparameters. The F1 score is a useful metric for evaluating classifiers especially when the distribution of classes is somewhat uneven, as is the case here. F-scores are generally calculated as the harmonic mean of the precision and recall scores. Precision refers to the ratio of true positives to all classified positives, and Recall refers to the ratio of true positives to all actual positives. A high precision score means a low rate of false positives, and a high recall score means a low rate of false negatives.

The F1 score is calculated as:

- 2 x Precision x Recall / (Precision + Recall)

In situations where the distribution of the target variable in a binary or multiclass classification task is unbalanced, the accuracy score becomes a less reliable indicator of the performance of a model. Given the somewhat unbalanced distribution of the target variable (roughly, 35% successful versus 65% unsuccessful), more emphasis will therefore be placed on the F1 score than on the accuracy score as a performance metric.

Further, given the size of the dataset, and the fact that the classification problem described in this project is quite realistic, I use the metric of **Training Time** to evaluate models. Training time is simply the time it takes to fit a machine learning model against the available training data, with lower training time being considered better than higher. In practical situations, one may choose a slightly less performant model over the highest-scoring model if it is orders of magnitude quicker to train.

II. Analysis

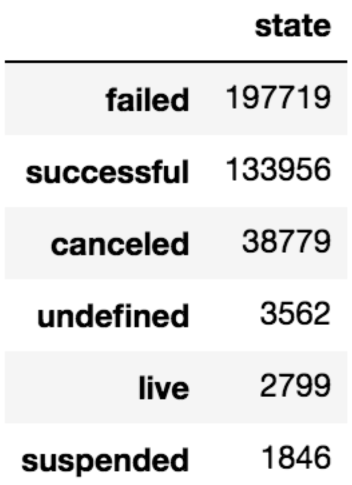
Data Exploration

The original dataset contains 15 features: 14 input features and one target feature. Of these, 6 have string values, 2 had date values, 2 had integer values and the remaining 5 have floating point values.

In this section we will describe interesting characteristics about the input and target features.

Target Feature: *Project State*

The target feature, *state*, has 6 distinct values, as seen in **Figure 2** below. 197,719 projects, or roughly **52.2%** of all projects failed, while 133,956 projects, or roughly **35.4%** of all projects were successful. Projects with the state 'live' will be excluded from the classification exercise[3], since a final determination cannot be made on their success value as yet.



	state
failed	197719
successful	133956
canceled	38779
undefined	3562
live	2799
suspended	1846

Figure 2: Distribution of Project States

Rows with state 'undefined' will be explored and adjusted based on whether or not pledges exceed goal values (and will be assigned 'successful' if so, and ‘failed’ if not). We will work on the assumption that the state was assigned 'undefined' due to an error in data capture, but that we can reassign it to desired state categories based on the target conditions.

Input Features

Some interesting summary statistics on the input features were observed:

- The mean project goal (in US dollars, using *usd_goal_real*) was **$45454.40**, though on comparison with the median goal of **$5500.00** it became clear that this figure was heavily skewed by extreme outliers, such as a project with a goal of US $1 billion which ended up receiving only $1 in pledges.

[3] The 'live' projects can perhaps be used as an extra test set for the final model in a further exercise, where the model predictions of this project can be tested on the real-world outcomes of these projects when they are concluded.

- The mean amount pledged (in US dollars, using *usd_pledged_real*) was **$9058.92**, though this again was highly skewed compared to the median pledge amount of **$624.33**. This would have been largely due to the 276 projects that raised over US $1 million, of which 5 projects raised over US $10 million.
- The number of backers per project was similarly skewed, with a mean of **105.62** backers, compared to a median of **12** backers, pulled up by outliers from the right tail of the distribution where 20 projects had over 50,000 backers. The project with the most backers had 219,382 backers.

I also explored the characteristics of the input features to prepare for the data preprocessing step.

- *main_category* was a candidate for one-hot-encoding, with 15 distinct main categories.
- *currency* was also a candidate for one-hot encoding, with 14 distinct values.
- *country* likewise, was likely to be one-hot encoded, with 23 distinct values.

Missing and Unusual Values

While a small number of missing values were found as a proportion of the original overall dataset, these missing values must be dealt with appropriately for the project to deliver meaningful outputs. Further exploration of the data uncovered some unusual properties, some of which coincided with missing values.

- 4 projects were found with missing values for the 'name' variable. This was surprising, as one would expect the name of a Kickstarter project to be mandatory for a project to go forward.
- 3797 projects were found with missing values for the 'usd pledged' variable. 3797 projects were also found with the country name *N,0”.* On further investigation, the projects with these unusual characteristics lined up exactly.
- Of the 3562 projects with state ‘undefined’, they all had missing values for ‘usd pledged’ and country value ‘N,0”’. Moreover, all these projects had what I termed ‘phantom backers’, i.e. they were reported to have 0 backers, though most of them had non-zero pledge values (by observing the *pledged* and *usd_pledged_real* values).

These highly unusual findings suggested to me that there were some errors either in the source data or in the process of obtaining the data. This would have implications for the data preprocessing step.

Exploratory Visualization

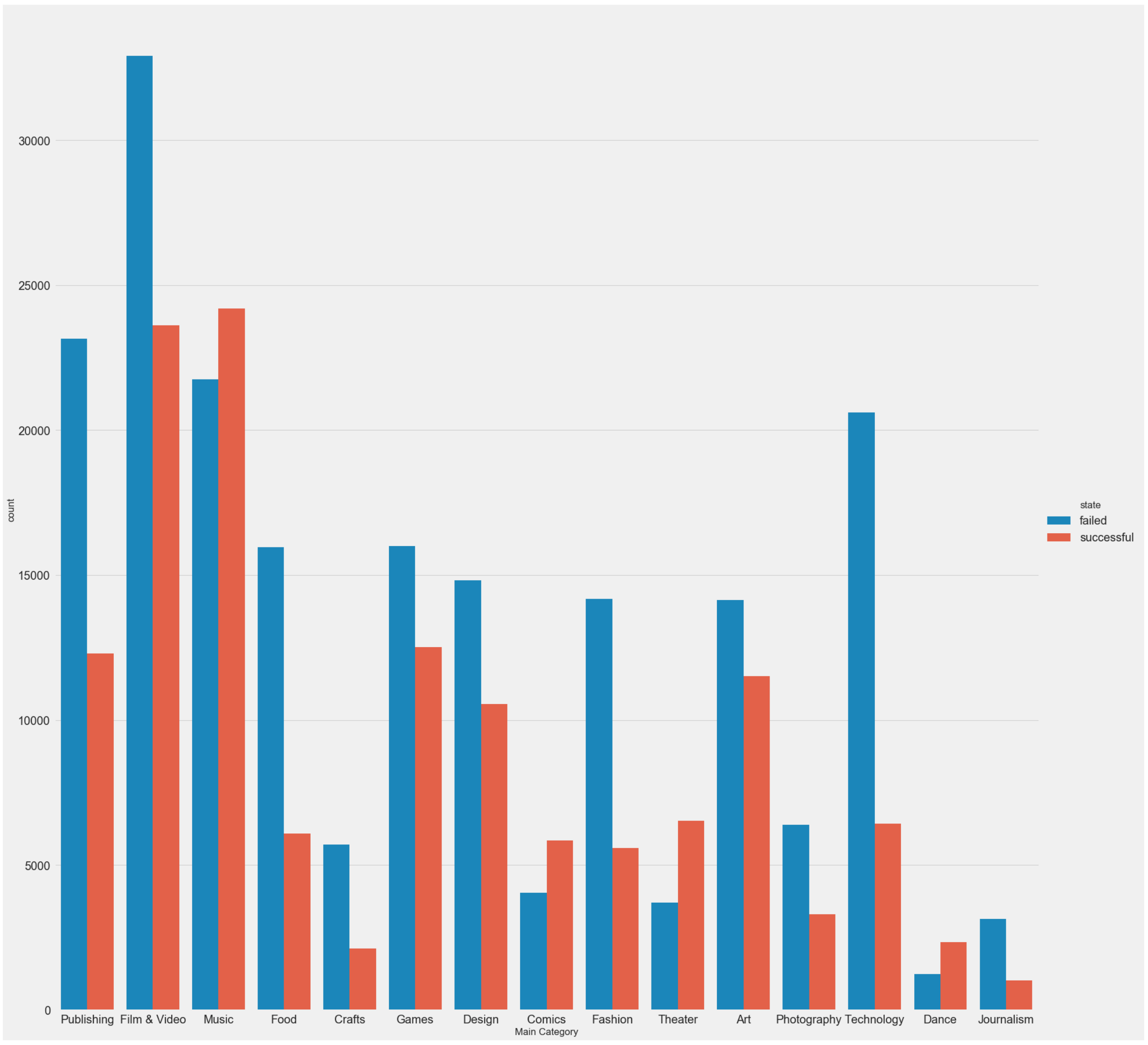


Figure 3: Successful and Failed Projects by Main Category

Figure 3 above shows the distribution of successful and failed Kickstarter projects, broken down by the main categories under which they are classified. From the chart, we can see that most categories have more failed projects than successful – which one would expect, since a majority of projects overall were failed. However, it is interesting to note that the *Music*, *Comics*, *Theater* and *Dance* categories had a greater number of successful than failed projects, while the *Technology* category appeared to have the highest ratio of failure to success.

Algorithms and Techniques

I will be using three models to classify successful and unsuccessful projects, namely:

i. Support Vector Classifier

ii. AdaBoost Classifier (with a Decision Tree base learner)

iii. XGBoost Classifier

Support Vector Classifier

The Support Vector Machines (or SVM) classifier is a powerful machine learning algorithm that attempts to create a decision boundary that maximises the margin between different classes of the target feature. For this reason, it is thought of as a large-margin classifier. The SVM aims to do this through its error term by punishing misclassified points that are far from the decision boundary much more than those that are close to the decision boundary.

SVMs can be tuned via a number of hyperparameters, including the C parameter, which is multiplied by the error term to determine the size of the margin between classes to be used, as well as the kernel, which is a function that changes the dimension of the feature space and allows for the decision boundary to be calculated. Given the simplicity of the dataset, I used the simplest kernel available for the SVM classifier, namely the linear kernel. I thought the SVM classifier would be appropriate to this task because of its ability to generate accurate predictions, and its ability to handle large numbers of features, which would be the case for the dataset after the data preprocessing stage was complete.

AdaBoost Classifier

AdaBoost is a boosted ensemble learning algorithm. What this means is that it combines multiple individual instances of a base ML model (known as "weak learners") to create a more powerful model (known as a "strong learner"). It does this by first running the weak learners on different subsets of the data, taking the misclassified points of each subset and assigning higher weights to them such that the value of the weighted sum of correct and incorrect points ends up being equal. The more information a model gives us, the higher the weight it will be assigned. This process continues iteratively according to the user's hyperparameter choices, until a strong learner - the sum of the weighted weak learners - is obtained. The default weak learner for AdaBoost is the Decision Tree classifier, which quickly and sequentially uses the decision boundary of the most descriptive features in a dataset to determine the correct value of the target feature.

I thought AdaBoost would be an appropriate model choice given the simple underlying feature set (14 input features), the binary classification task to which the Decision Tree classifier is suited, and the fact that the algorithm is relatively quick to train. My working assumption was that AdaBoost would perform adequately even after the creation of new features and the expansion of feature space introduced by one-hot encoding.

XGBoost Classifier

The final model to be considered was XGBoost (eXtreme Gradient Boosting). XGBoost is an implementation of the gradient boosted decision tree algorithm. Like AdaBoost, this model uses Decision Trees as the base learner. This introduction to XGBoost states that "[g]radient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. It is called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models." XGBoost is known for performing especially well on tabular data, which is how the dataset for this project is structured.

Over the past few years, XGBoost has come to dominate Kaggle competitions and is known for a very strong combination of performance and training speed. Since I use both performance metrics and training time to evaluate models for this project, I therefore thought this would be a highly appropriate model choice. Moreover, I wanted to use at least one model that was not explored in the ML Nanodegree, to give myself the practice of learning about and working with different models, and comparing them to what I have learned.

Benchmark: Logistic Regression

The benchmark to which I will compare the performance of the above three classifiers is a logistic regression model. Logistic regression is one of the simplest but still most widely used classifiers in ML, especially for smaller, less complex datasets. The model outputs the probability that a given input belongs in a particular class of the target feature.

As referred to in my Capstone Proposal, this Kaggle kernel used logistic regression on the Kickstarter dataset to deliver an accuracy score of **63.19%**, and I intended to use this as the benchmark for the project. However, when I ran my own logistic regression on the dataset, the model reported an accuracy score of **90.68%** and F1 score of **0.88**, with a training time of 2.04 seconds. I therefore used the higher scores of this model as the benchmark.

III. Methodology

Data Preprocessing

To be processed by the classification models we will be using in this project, the observations for the input and output features must be converted to numerical values, through **label encoding**, which converts the string values of the categorical observations to corresponding numerical values, and **one-hot encoding**, which creates a distinct binary numerical feature for each category observed in the original feature.

Label Encoding

For the target feature, *state*, I encoded successful projects with a value of 1, and failed projects with a value of 0.[4] Prior to this, projects with state ‘undefined’ were re-assigned a state based on the relationship between their goal and their pledged values. If the pledges were greater than or equal to the goals, I assigned state ‘successful’ and if pledges were less than goals, I assigned ‘failed’.

All other project states were dropped from the dataset, i.e. projects with the states ‘live’, ‘canceled’ or ‘suspended’ were not included, so as to reduce the problem to one of binary classification, as specified in the **Problem Statement.**

One-Hot Encoding

From the **Data Exploration** section, all the candidate features for one-hot encoding, namely *country*, *main_category* and *currency* were processed in this manner.

Originally, I dropped the *category* feature from the dataset, as there were so many distinct values (159) that I thought encoding them would increase the feature space to the point where the dataset becomes unwieldy and model training times would increase significantly. I also thought that enough of the information contained in *category* was captured in the *main_category* feature. However, on experimenting with training the models I found that model performance, especially that of the linear Support Vector Classifier (SVC) improved significantly when *category* was included and one-hot encoded. Therefore, the *category* feature is also one-hot encoded in the final dataset.

Feature Engineering

Given the low dimensionality of the original dataset (14 input features), I constructed some new features from the given features, with the intention of discovering interesting variables that would have explanatory power on the target feature, and thereby improve the performance of the trained models.

The new features I created were:

- **Project Name Length** (*name_length)*: the length of the string object *name* for each project.
- **Uppercase Characters in Project Name** (*caps_in_name)*: the number of uppercase characters in the string object *name* for each project.
- **Campaign Length** (*cam_length)*: the length of the campaign in days. This required converting the *launched* and *deadline* features to comparable date/time format then subtracting the deadline date from the launch date.

[4] Originally, I encoded suspended and cancelled projects (since they were also unsuccessful) as 0, but later decided against this as I thought exogenous factors (outside of the scope of this dataset and project) would have accounted for projects taking up such states. This alternative specification had little impact on the performance of the models, so there was little hesitation in using the more parsimonious specification.

The intuition behind *name_length* and *caps_in_name* was that these may be somewhat of a proxy for the quality of a project. In the absence of project description data beyond the name and category features in the dataset, the assumption was that longer names with more uppercase characters would be of potentially lower quality than those with more understated titles. Moreover, I needed numerical representations of the project name, since machine learning models cannot work with string objects such as *name*.

For *cam_length*, the intuition was that campaigns of greater length would have a greater chance of success than shorter ones on average.

Missing and Unusual Values

Given the observation of missing and unusual values in the **Data Exploration** phase, I came up with a strategy to treat with them.

- The 4 projects with missing name values were dropped from the dataset. Because there were so few of these as a proportion of the dataset, I judged that this would not cause undue harm to the final output of the models.
- I removed the column '*usd pledged'* entirely. This column featured 3797 missing values. One option to treat with the missing values in 'usd pledged' would have been to fill them in using the *fillna()* function in *pandas*. However, the column '*usd pledged*' overlaps significantly with *usd_pledged_real*, for which there were no missing values, and consequently little harm is done by removing the former column entirely.
- Finally, I decided to drop rows with strange values such as *country* = 'N,0"' with phantom backers, since they were less than 1% of the dataset and probably reflected some errors in data collection. I did not want these unadjustable anomalies to affect model performance.

Feature Selection

I removed a number of features that were in the original dataset, either due to their lack of explanatory power, non-numerical format, or the fact that they would

- After creating numerical features from the *name* feature and creating a new *cam_length* feature in the **Feature Engineering** phase, I dropped the *name*, *deadline* and *launched* features from the dataset.
- I dropped *ID* from the dataset, since it was unlikely to have any explanatory power over the success or failure of projects.
- I dropped *pledged*, *usd_pledged_real* and *goal* as successful projects can be found by subtracting the relevant goal from the relevant pledge value; this was thought to make the problem trivially easy to solve, and this is not the goal of the exercise.

Conclusion

The dataset began with **378,661** observations on **15** features, and after the data preprocessing step ended up with **331,462** observations on **216** features.

Implementation

The input features (X) were separated from the target feature (y), and the data was split into training and testing sets, with 20% of the data (66,293 samples) allocated to the test set. In the initial implementation, all four models (benchmark and three candidate models) were trained using random initialisation (*random_state* = 42) for reproducibility.

To carry out the implementation of these models, I used the following tools:

- **Scikit-learn**, version 0.18.1
- **xgboost**, version 0.72
- *LogisticRegression* from **sklearn.linear_model**
- *LinearSVC* from **sklearn.svm**
- *AdaBoostClassifier* from **sklearn.ensemble**
- *Accuracy_score, F1_score* and *make_scorer* from **sklearn.metrics**
- *GridSearchCV* and *train_test_split* from **sklearn.model_selection**

In the initial implementation, I specified hyperparameters as described below.

SVM

I specified the C parameter at its default value of 1, as variations such as 0.01, 0.1, 0.5, 0.7, 1.5 did not add to performance and in some instances severely reduced it. However, the *max_iter* parameter gave the model improved performance with a value of 1500, compared to the default of 1000 and other values that were tried such as 500, 700, 2000 and 2500.

AdaBoost

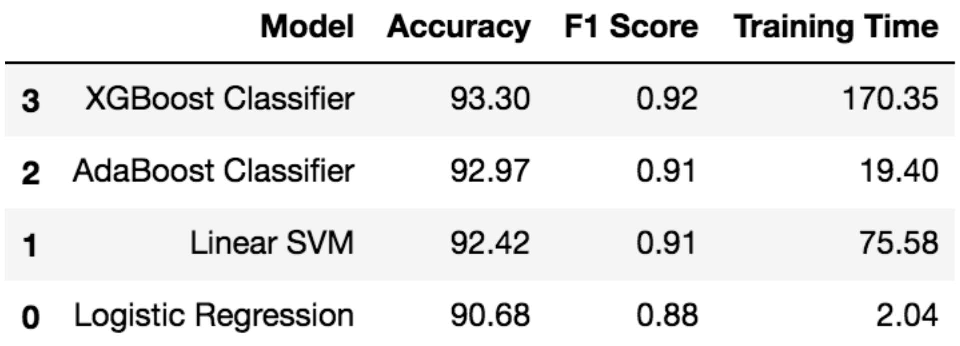
I used an AdaBoost model with a Decision Tree base estimator. For the base Decision Tree, I chose a *max_depth* value of 1 for the initial specification. In other words, the base estimators/weak learners would only have one level of branching – such Decision Trees are also known as *decision stumps*. I also experimented with varying other Decision Tree parameters such as the *criterion* function, which is the function that determines the quality of a tree split – though the alternative option of 'entropy' for information gain did not deliver better results than the default of Gini impurity. I tried varying the *max_features* parameter, which specifies the number of features to consider when choosing a tree split, however these variations, i.e. using the square root of the total number of features, or using the base-2 log of the number of features, significantly worsened performance, so the default value (*max_features* = *n_features)* was maintained.

For the overall model, I used the default value of *n_estimators*, set to 50 – this specifies the maximum number of weak learners to be used in the ensemble model. I also used a *learning rate* of 0.8, which gave good performance. The learning rate is a scaling/discount factor for the output of each weak learner, and there is an inverse relationship between *n_estimators* and *learning_rate*.

XGBoost

I experimented with varying the *learning rate*, which functions similarly to that of AdaBoost, and this generated some positive outcomes. After trying specifications of 0.1, 0.2 and 0.3, the model with learning rate 0.1 generated the best results.

Figure 4 below summarises the performance of the three chosen models as well as the benchmark model.



	Model	Accuracy	F1 Score	Training Time
3	XGBoost Classifier	93.30	0.92	170.35
2	AdaBoost Classifier	92.97	0.91	19.40
1	Linear SVM	92.42	0.91	75.58
0	Logistic Regression	90.68	0.88	2.04

Figure 4: Model Comparison

Based on the performance metrics of accuracy score and F1 score, the **XGBoost** classifier was judged to be the best model to task. XGBoost performed very slightly better with regard to accuracy and F1 score than AdaBoost. However, this came at the cost of taking roughly 9 times as long to train as AdaBoost. The SVM with a linear kernel underperformed the two boosted tree models, with accuracy and F1 scores below those of AdaBoost and XGBoost, and a training time between those of the AdaBoost and XGBoost models.

All three candidate models outperformed the logistic regression benchmark in terms of accuracy and F1 score, though they both took much longer to train than the extremely rapid benchmark. Because they were so close in performance, I decided to try to refine both XGBoost and AdaBoost by tuning key parameters, then compared their optimised performance to settle upon the very best model. Due to limitations on my local machine, only a very simple tuning of the XGBoost parameters could be completed. The results of the model tuning process are presented in the following section.

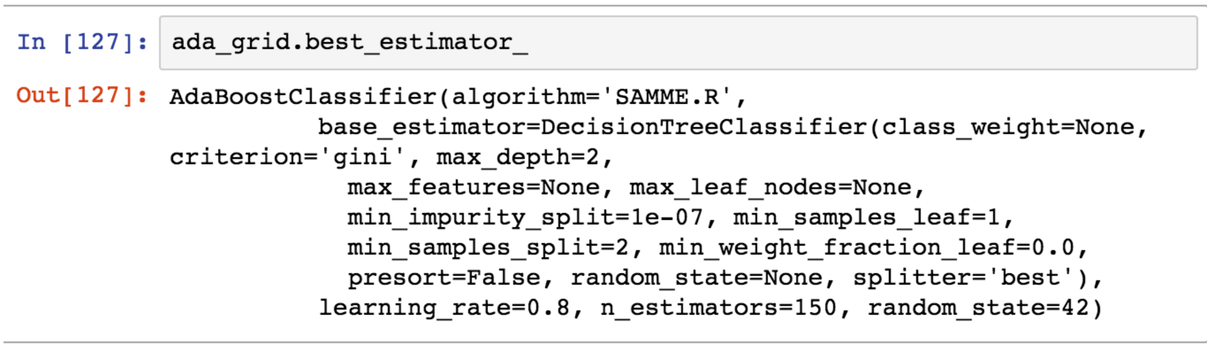
Refinement

I tuned both the AdaBoost and the XGBoost models using Grid Search cross validation. Grid Search cross-validation tests different parameter settings of a machine learning model to come upon the best combination according to a chosen evaluation metric, in this case the F1 score.[5]

For the AdaBoost classifier, the parameters that I sought to optimise with Grid Search were as follows:

- *n_estimators*, which chooses the maximum number of base estimators that the algorithm will use to boost its performance on the classification task. The default value of *n_estimators* for AdaBoost in the Python *scikit-learn* library is 50, so I sought to improve performance by trying specifications of 100, 150 and 200 base estimators.
- *base_estimator__max_depth*, which selects the maximum tree depth of AdaBoost's underlying Decision Tree estimator. In the initial implementation phase, I elected to use the simplest specification of *max_depth* = 1, but when optimising with grid search, I tried multiple settings between 2 and 8.

The output of Grid Search showed that the best AdaBoost model featured 150 base estimators, a base estimator *max_depth* of 2 and (based on the initial specification) a learning rate of 0.8, as shown in the snippet below (**Figure 5**).

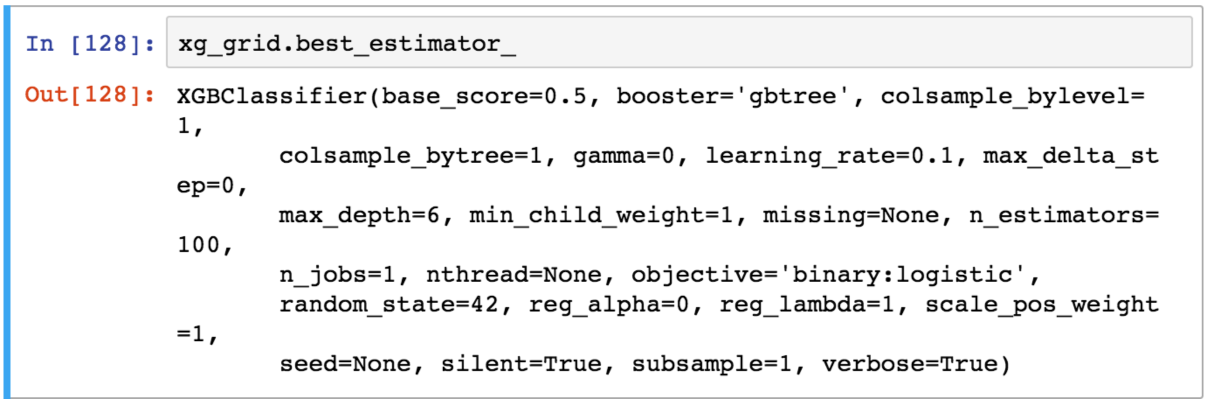


```
In [127]: ada_grid.best_estimator_

Out[127]: AdaBoostClassifier(algorithm='SAMME.R',
                    base_estimator=DecisionTreeClassifier(class_weight=None,
          criterion='gini', max_depth=2,
                      max_features=None, max_leaf_nodes=None,
                      min_impurity_split=1e-07, min_samples_leaf=1,
                      min_samples_split=2, min_weight_fraction_leaf=0.0,
                      presort=False, random_state=None, splitter='best'),
                    learning_rate=0.8, n_estimators=150, random_state=42)
```

Figure 5: Optimised AdaBoost specification

For the XGBoost classifier, I was limited in the amount of tuning I could do, but was able to optimise over the *max_depth* parameter. I tried multiple values between 1 and 8. The output of Grid search showed that the best XGBoost model featured a base estimator *max_depth* of 6, with a learning rate of 0.1, as shown in the snippet below (**Figure 6**).

[5] For the XGBoost classifier, I attempted to optimise the value of *max_depth*, which sets the maximum depth of the base decision tree, largest length between the root (beginning) and a leaf(endpoint). Large depth very often causes overfitting, since a tree that is too deep, can memorize the data. Small depth can result in a very simple model, which may cause underfitting. The default value of *max_depth* in XGBoost is 6, and I tested specifications of 4 and 8, however they failed to return results after over an hour of running.



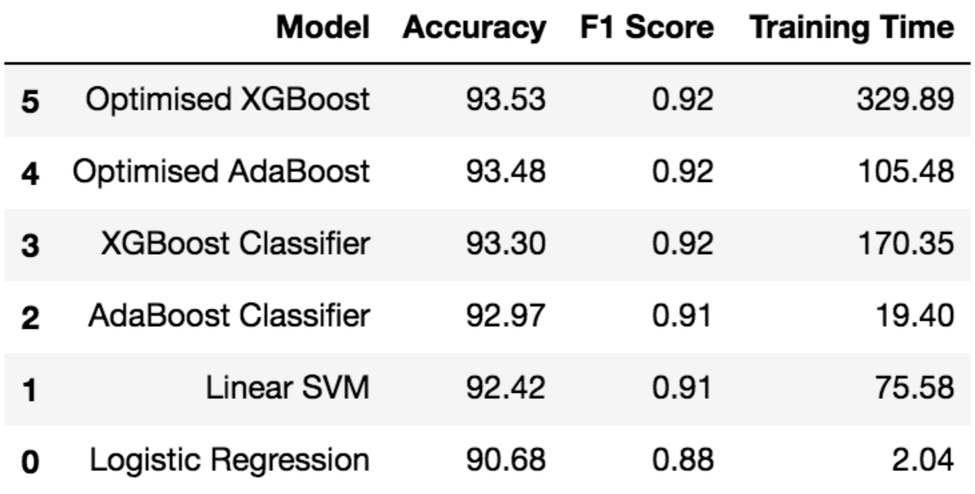
```
In [128]: xg_grid.best_estimator_

Out[128]: XGBClassifier(base_score=0.5, booster='gbtree', colsample_bylevel=
          1,
                 colsample_bytree=1, gamma=0, learning_rate=0.1, max_delta_st
          ep=0,
                 max_depth=6, min_child_weight=1, missing=None, n_estimators=
          100,
                 n_jobs=1, nthread=None, objective='binary:logistic',
                 random_state=42, reg_alpha=0, reg_lambda=1, scale_pos_weight
          =1,
                 seed=None, silent=True, subsample=1, verbose=True)
```

Figure 6: Optimised XGBoost specification

IV. Results

Model Evaluation and Validation

The performance of the models, including the optimised AdaBoost model, is presented below in **Figure 7.**



	Model	Accuracy	F1 Score	Training Time
5	Optimised XGBoost	93.53	0.92	329.89
4	Optimised AdaBoost	93.48	0.92	105.48
3	XGBoost Classifier	93.30	0.92	170.35
2	AdaBoost Classifier	92.97	0.91	19.40
1	Linear SVM	92.42	0.91	75.58
0	Logistic Regression	90.68	0.88	2.04

Figure 7: Model Comparison, Including Optimised AdaBoost and XGBoost

As the table shows, the optimised AdaBoost runs almost exactly the same as the optimised XGBoost classifier in terms of accuracy and F1 score, while taking less than half the time to train. Therefore, taking both the performance-time trade-off and my difficulties in optimising XGBoost into account, I settled upon **Optimised AdaBoost** as the best model for this project.

The hyperparameters of this optimised AdaBoost model are as follows:

- *n_estimators:* 150, compared to the default of 50
- *learning_rate:* 0.8, compared to the default of 1 – reflecting the inverse relationship between *n_estimators and learning_rate*
- *Base estimator: Decision Tree*

- *Base estimator max_depth*: 2, compared to the default of 1

The specification of the optimised AdaBoost model can therefore be described as slightly more complex than the initial specification, accounting for both increased performance and increased training time, while narrowing the gap to XGBoost in performance while maintaining a significant advantage in training speed.

Justification

As **Figure 7** above shows, all of the candidate models performed better than the benchmark logistic regression model. The optimised AdaBoost classifier is joint in first place with XGBoost on F1 and accuracy score. Admittedly, the benchmark model performs very well to task, and the best models are only a few points better while taking orders of magnitude longer to train.

V. Conclusion

Free-Form Visualization

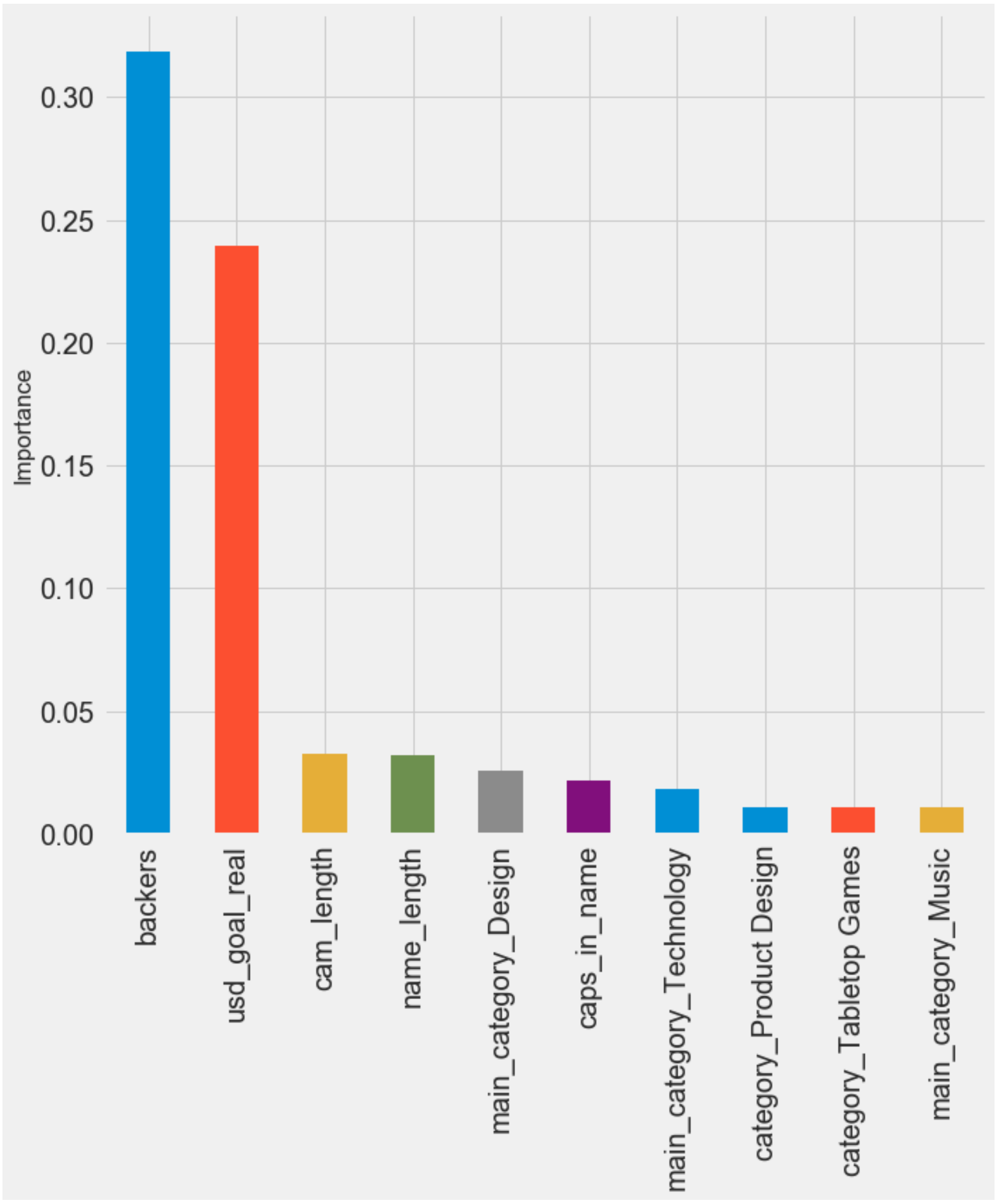


Figure 8: Feature Importances

Plotting the most important 10 features for the optimised AdaBoost model, in **Figure 8**, threw up some interesting results. Some machine learning models, especially those based on decision trees - including AdaBoost and XGBoost - are able to report on feature importances, the relative importance of each feature with respect to the predictability of the target variable.

First, the number of backers of a project had the biggest contribution to explaining success or failure of the project, with a score of **0.32.** The project goal (in US dollars) was the second most important feature, with a feature importance score of **0.32**. The campaign length and name length feature (*cam_length and name_length*) had more impact than the one-hot encoded categorical features, but somewhat less importance than expected, with only **0.03** importance scores each. The *caps_in_name* feature had little impact as well, with only **0.02** as its score.

Reflection

Overall, this project took a dataset from a public repository (Kaggle), explored it for interesting characteristics, cleaned it through a variety of data preprocessing steps, then ran it through four classification models (three candidate models and a benchmark model), using three evaluation metrics to compare their performance.

Data preprocessing took the longest time in this project. That being the case, I was happy to spend this time on data preparation because in the MLND projects we worked with either clean datasets or got a lot of help in preprocessing steps for the course projects. I enjoyed the experience of working on this phase independently, especially as data cleaning/preprocessing is frequently cited as an important and time-consuming step in the data science/ML practitioner's real-world experience.

I was pleasantly surprised at the accuracy of my trained models in comparison compared to those I observed from looking at Kaggle kernels. I take that as a sign that my efforts in data preprocessing had a positive impact on performance. Further, I think all of the models I selected were appropriate to the task, as they all outperformed the benchmark model on performance, though not on training time.

Improvement

Finally, I think there were some areas where I could have improved the implementation of this project to derive better results. Given more processing power (or, had I deployed my project on an Amazon EC2 p2.xlarge instance), I would have liked to optimise the XGBoost classifier to derive even better results. Similarly, a multi-layer perceptron algorithm may have been well-suited to this task, though I did not explore this avenue. Further, I suspect there are some more creative and potentially performance-enhancing features that could be created from the original dataset, beyond the three I engineered, as I was somewhat disappointed with their impact in relation to feature importance. While I was quite impressed with the performance of my final solution, especially in comparison to the Kaggle kernels I observed while preparing this project, I think a better solution could potentially be designed.