Kickstarter Project Success Analysis

Steve Bachmeier
2018-12-13

```
# Code to run for report

#--------------------------------------------------------------
# Run the following two lines to hide the In[] and Out[] margin.
# Doing so will not allow headings to be collapsed.

from IPython.core.display import display,HTML
display(HTML('<style>.prompt{width: 0px; min-width: 0px; visibility: collapse}</style>'))

#--------------------------------------------------------------
# Run the following two lines to re-load the df_results dataframe
import dill
df_results = dill.load(open("df_results.pkl", "rb"))
```

1 Synopsis

Several machine learning models were trained on a cleaned training set (60% of the entire dataset) and tested on a cleaned test set (20% of the entire dataset) - the results of these models are shown in the table below.

An optimized random forest model provides a decent **10-fold cross validation mean accuracy of 71.6% while also featuring a (qualitatively) quick run-time**.

The specific random forest model used features 100 trees, Gini impurity criterion, the maximum number of features to consider for a split equal to the square root of all of the features, and the minimum number of samples required to be at a leaf node of 25 (ie *n_estimators*=100, *criterion*="gini", *max_features*='sqrt', and *min_samples_leaf*=25).

```
df_results
```

	model	time_fit	time_predict	time_10_fold_CV	accuracy	acc_10_fold
0	Naive Bayes	0.101767	0.036270	1.53579	0.623551	0.624725
1	Logistic Regression	1.822522	0.009085	17.0696	0.694333	0.689984
2	K Nearest Neighbors	7.340739	101.319590	344.234	0.682739	0.675807
3	SVM, Linear	954.375944	90.726687	None	0.677168	None
4	SVM, RBF	1091.777649	120.427602	None	0.682908	None
5	Decision Tree	0.437127	0.014542	4.82765	0.664357	0.662544
6	Random Forest (10-fold)	1.016743	0.104582	10.463	0.683248	0.680784
7	PCA (n=2), Naive Bayes	0.024343	0.003595	0.284533	0.609355	0.608927
8	Random Forest (Optimized)	5.856856	0.470576	66.9746	0.718257	0.716067
9	Logistic Regression (Optimized)	6.454474	0.003956	68.8582	0.695662	0.691059
10	KNN (Optimized)	7.253727	48.397783	211.557	0.700328	0.69877

General trends were not so easy to recognize with the exception of the influence of the variable *staff_pick*; *staff_pick* is the predictor most highy correlated with *launch_state*:

- *staff_pick* - *launch_state* correlation: 25%
- 53% of projects without *staff_pick* succeed
- 89% of projects with *staff_pick* succeed

From https://www.kickstarter.com/blog/how-to-get-featured-on-kickstarter (https://www.kickstarter.com/blog/how-to-get-featured-on-kickstarter), it appears as if projects are featured when they catch the eye of the Kickstarter staff via creativity, a nice and visually appealing site, etc. ie, they are **not** just picked due to them being funded well. **Projects that are featured on Kickstarter (ie *staff_pick* = 1) have a high correlation with success, although it is difficult to tell if featured projects are already on track to be fully funded.**

2 Overview

2.1 Background

Having spent six years as a mechanical engineer in the silicon valley where ideas are big but funding is small, I've always been intrigued by the concept of crowd-funding. As an end user/backer, however, we want to maximize the chances that the projects we back actually successfully launch. This project uses historical data from the popular project-launching website Kickstarter to look for trends and make predictions about whether a project is likely to be successfully funded or not.

2.2 Data

The raw Kickstarter data (the JSON file updated at 2018-10-18) was downloaded from: https://webrobots.io/kickstarter-datasets/ (https://webrobots.io/kickstarter-datasets/). It is assumed that this data is accurate and no attempt was made to verify the web scraping tools used.

Notes from raw data downloaded:

- **From April 2015 we noticed that Kickstarter started limiting how many projects user can view in a single category. This limits the amount of historic projects we can get in a single scrape run. But recent and active projects are always included.**
- **From December 2015 we modified the collection approach to go through all sub-categories instead of only top level categories. This yields more results in the datasets, but possible duplication where projects are listed in multiple categories. Also from December 2015 JSON file is in JSON streaming format. Read more about it here: https://en.wikipedia.org/wiki/JSON_Streaming (https://en.wikipedia.org/wiki/JSON_Streaming)**
- **We receive many question about timestamp format used in this dataset. It is unix time. Google has a lot of information about it.**
- **Files are compressed, size in area of 100mb. Uncompressed size around 600mb.**

Note that no attempt was made for this project to ensure we have the entirety of the project history. Also, due to Github size constraints, the raw dataset is not uploaded to this repository.

2.3 Goal

There are two potential goals of this project:

1. Analyze the raw data obtained to look for any interesting trends.
2. Build a prediction algorithm to try and predict whether future projects will successfully launch.

3 Model creation

This section outlines the analysis completed. Refer to Appendix A1 for relevant code.

3.1 Data preparation

Refer to appendix A1.1 Data preparation for code.

The downloaded dataset from https://webrobots.io/kickstarter-datasets/ (https://webrobots.io/kickstarter-datasets/) came in JSON format; each of the 205,696 rows representing a project's details that were wrapped up in a serialized set of dictionaries and nested dictionaries. Unpacking this file was not trivial - the summary of the process is as follows:

1. Open the file with utf8 encoding and load each line to a new object.
2. The raw object includes four columns of dictionaries of which only the *data* column is relevant; extract that *data* column.
3. Convert the json file to a Pandas dataframe.
4. Unpack each dictionary with *json_normalize()*. Note that this does not unpack columns with NaN values.
5. Unpack remaining dictionary columns (those with NaN values) manually by applying the Pandas *Series()* method.
6. Concatenate the newly unpacked columns to the dataframe.
7. Drop the original json columns from the dataframe.
8. Split the dataframe into a working set (later to be the train and test set) and a validation set. We use a random 20% smapling of the entire raw dataset for the validation set.

```
X, X_v, y, y_v = train_test_split(df_raw.drop(columns=['state']),
                                  df_raw['state'], test_size=0.2,
                                  random_state=101)
```

The working and validation sets now consist of 97 columns (where each columns is a different variable with one of them being the outcome).

3.2 Data cleaning

Refer to appendix A1.2 Data cleaning for code.

With the raw data now in a usable dataframe, we can clean it up for machine learning use.

3.2.1 Clean up columns

The first step is to drop clearly useless variables. This demands some amount of reasoning. For example, it is perhaps obvious that a project photo urls, creator avatar photos, and creator profile blurbs are not useful for using machine learning to make predictions. However, other variables may not be so obvious. For example, a creator's name could be used to identify the gender (which is not provided directly in the dataset) which in turn might shed some light on project success (note that for this analysis I did indeed drop the creator's name from the dataset).

One interesting variable that took special consideration is *profile_state*. Digging into it showed that there are only two unique values for *profile_state*: 'active' and 'inactive'. Further, only 11.7% of the project profiles are labeled 'active'. It is assumed that projects go 'inactive' after a certain period of latency and so cannot be used in predicting project success (ie even a successfully funded project profile may go active after some amount of time past the deadling). I decided to drop *profile_state*.

Another tricky one was *usd_type*. Frankly, I was unable to get a firm grasp on what exactly it is. There were many instances of a project country being labeled, say 'US' with it's currency being 'USD' but then *usd_type* being 'international'. Further, there were instances of empty values. Finally, the vast majority of *usd_type* is labeled international. I decided to drop it.

Other nonobvious variables that I deleted include: *name*, *blurb*, *loc_state* (too granular), *location_country* (largely redundant with *loc_country* but far more granular), *currency* (the pledges are all in USD), *currency_trailing_code*, and *state_changed_at*.

Once the useless columns were dropped, I renamed *category_slug* to *category* and *state* to *launch_state*.

Finally, I reordered the columns into a more intuitive order including putting *launch_state* first.

3.2.2 Extract categories

The *category* column (initially labeled *category_slug*) included primary categories and sub-categories in the format 'primary_category/sub_category', eg 'art/painting' and 'comics/webcomics'. I simplified the *category* variable by extracting the first word, eg 'art/painting' became 'art' and 'comics/webcomics' became 'comics'.

3.2.3 Drop duplicate rows

There were a fair amount of duplicate rows which are easy to remove with `df.drop_duplicates(inplace=True)`. However, even after this, the dataset included rows that were mostly duplicates with the exception of just a few column values. In order to keep the dataset tidy (where each row is a unique observation or, in this case, project), I had to remove any rows with duplicate project IDs. I decided to, in the case of duplicate IDs, keep those with the highest *pledged* value (assuming that this was input after the other rows and so is more accurate). This was accomplished with `df = df.sort_values('pledged', ascending=False).drop_duplicates('id').sort_index()`. These two steps resulted in no duplicate ID values and so a tidy dataset.

3.2.4 Convert relevant values to datetime

At this point, *deadline* and *launched_at* contained string values that are in the unix timestamp format. These variables were converted to datetime via

```
df['deadline'] = df['deadline'].apply(datetime.utcfromtimestamp)
df['launched_at'] = df['launched_at'].apply(datetime.utcfromtimestamp)
```

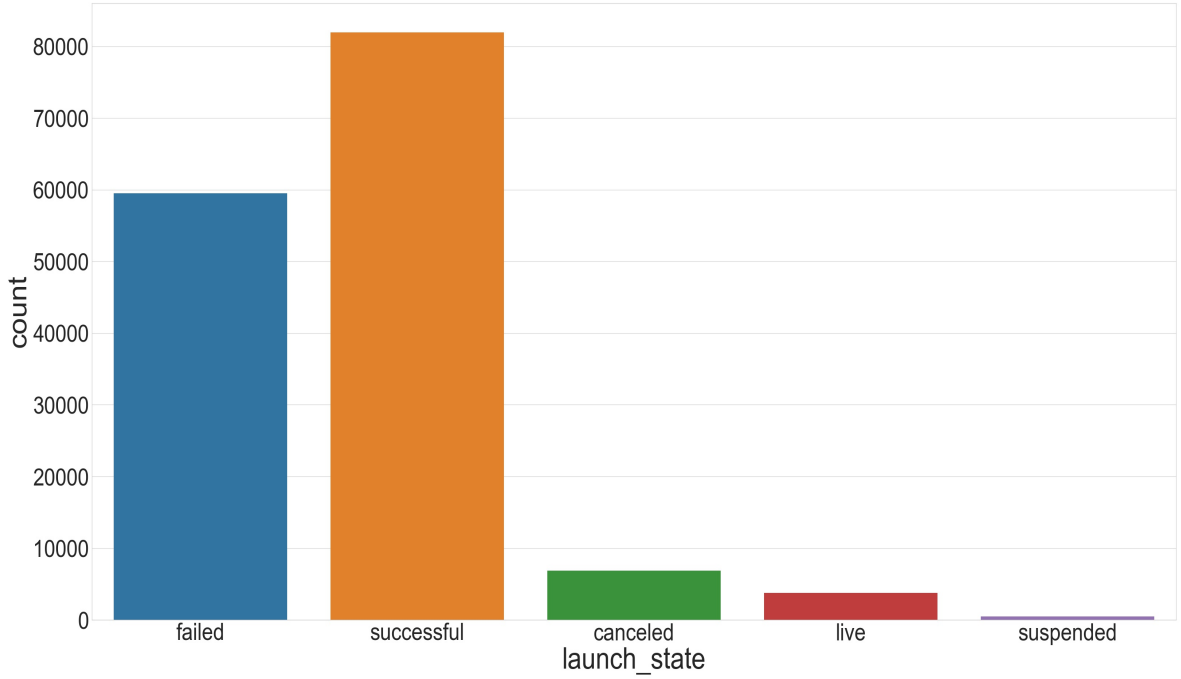
3.2.5 NA / Null / empty value imputation

At this point the dataframe was completely clean of any NA, Null, or empty values. This was easily checked with

```
if df.isnull().sum().sum() != 0:
    print('*** WARNING: There are null values ***')
if df.isna().sum().sum() != 0:
    print('*** WARNING: There are NA values ***')
if (df=='').sum().sum() != 0:
    print('*** WARNING: There are empty string (\'\') values ***')
```

3.2.6 Clean up the outcome variable *launch_state*

There are five values for *launch_state* (previously *state*): 'failed', 'successful', 'canceled', 'live', and 'suspended'.



For the purposes of this project, it makes sense to keep only 'failed' and 'successful' projects (since those labeled 'canceled' and 'suspended' cannot be backed to begin with and those labeled 'live' are exactly the projects we are trying to predict). We thus query only *launch_state* values of 'failed' and 'successful' (and type None just for completeness).

```
df.query("launch_state == 'failed' | "
         "launch_state == 'successful' | "
         "launch_state == None", inplace=True)
```

3.2.7 Create dummy variables

Of the remaining variables, *category* and *country* are categorical, ie they are labels rather than numbers. For the machine learning algorithm, I needed to convert these to dummy variables:

```
category = pd.get_dummies(df['category'], drop_first=True)
country = pd.get_dummies(df['country'], drop_first=True)
```

Note that I did drop the first dummy variable to ensure the correct degrees of freedom.

3.2.8 Convert remaining string variables to integers

There were still several variables with categorical binary values that could be converted to binary integers.

- *launch_state*: ['failed', 'successful'] should be [0, 1]
- *staff_pick*: [False, True] should be [0, 1]
- *spotlight*: [False, True] should be [0, 1]

I created a dictionaries to define what the string should be converted to and then mapped it to the relevant variables.

```
d_launch_state = dict(zip(['failed','successful'], range(0,2)))
launch_state = df['launch_state'].map(d_launch_state)

d_staff_pick = dict(zip([False,True], range(0,2)))
staff_pick = df['staff_pick'].map(d_staff_pick)

d_spotlight = dict(zip([False,True], range(0,2)))
spotlight = df['spotlight'].map(d_spotlight)
```

3.3 Variable reduction

At this point we have a tidy dataframe with 141,447 rows and 46 variables (most of which are dummy *country* and *category* variables). We can now look into paring down the number of variables to help reduce over-fitting of the machine learning algorithm(s).

3.3.1 Zero variance

We start by checking for zero variance variables, ie those variables that do not change at all throughout the entire dataset. Note that we've already removed some of these during the data cleaning phase, but it's still a good check.

```
sel = VarianceThreshold(threshold=0.0)
sel.fit_transform(X=df.drop(columns=info_variables)).shape[1] - df.drop(columns=inf
o_variables).shape[1]
```

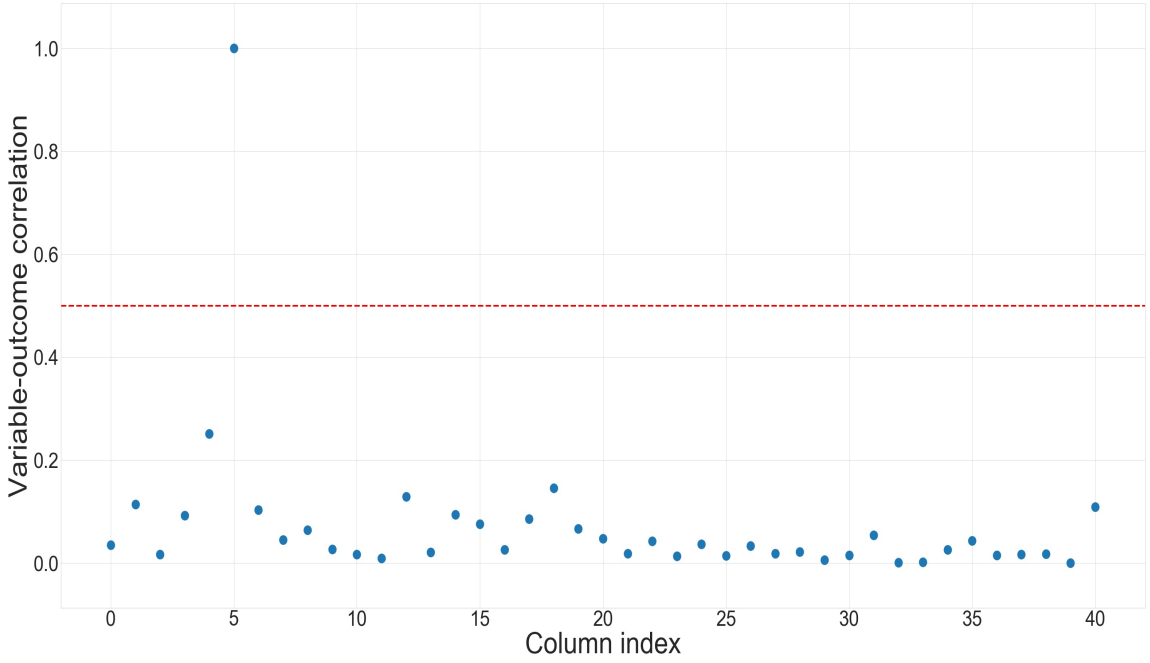
There were no zero variance variables.

3.3.2 Near-zero variance

It was decided not to search for or remove near-zero variance variables. One reason is that many of the dummy variables will certainly have variances near zero, eg a particularly small *country* may only appear a hand full of times in the entire data set and so the vast majority of values will be 0. Further, there is evidence that near-zero variance variables can still have a significant impact on the outcome.

3.3.3 Variable - outcome correlation

Variables with a very high correlation to the outcome we are trying to predict should be given extra consideration and possibly dropped. For this analysis, I use a threshold correlation of 0.5 - a single variable is found as shown in the plot below. Note that the dashed red line is the threshold value.



It turns out that this single variable of interest is *spotlight* which, as shown in the plot, has a perfect correlation of 1 with *launch_state*. In other words, it's a perfect predictor (which obviously seems suspicious). From https://techcrunch.com/2015/03/25/kickstarter-spotlight/ (https://techcrunch.com/2015/03/25/kickstarter-spotlight/), we see that spotlight happens for successfully funded projects and acts as a way to update the project time line. It clearly does nothing in helping predict funding success; I dropped it.

The next highest correlation is *staff_pick* at 0.25, well under the threshold.

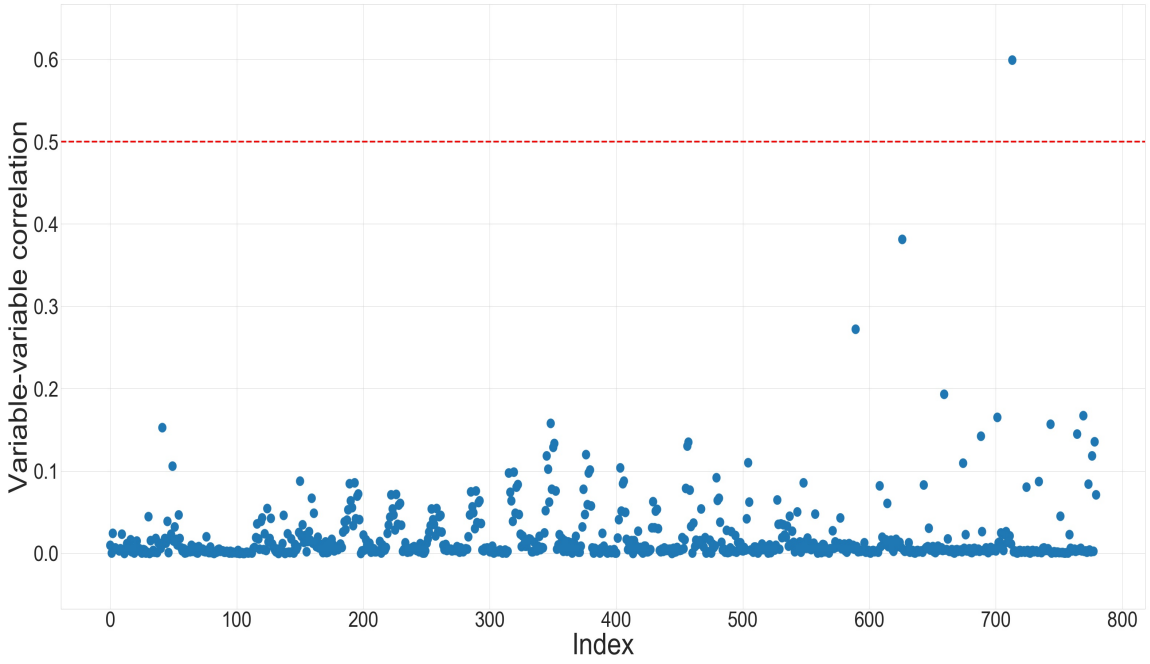
3.3.4 Variable-variable correlation

Next we consider multicollinearity, ie where a variable can be predicted by other variables. One way to battle this is by ensuring all variables have a variable-variable correlation beneath some threshold. For this analysis, we assume this threshold is 0.5.

The trick here is to create an upper correlation matrix with the ones diagonal removed, unstack it, sort the values in descending order, and filter by all correlation values greater than the threshold.

```
corMat_upper = corMat.where(np.triu(np.ones(corMat.shape), k=1).astype(np.bool))
corMat_upper.unstack().sort_values(kind='quicksort')[corMat_upper.unstack().sort_va
lues(kind='quicksort') > .5]
```

The result is that a single variable pair has a correlation larger than 0.5:



The variable pair in question is *US* - *GB* and has a correlation of 0.599. It does not make sense to drop a country just because it correlates with another country and so we keep both *US* and *GB* dummy variables.

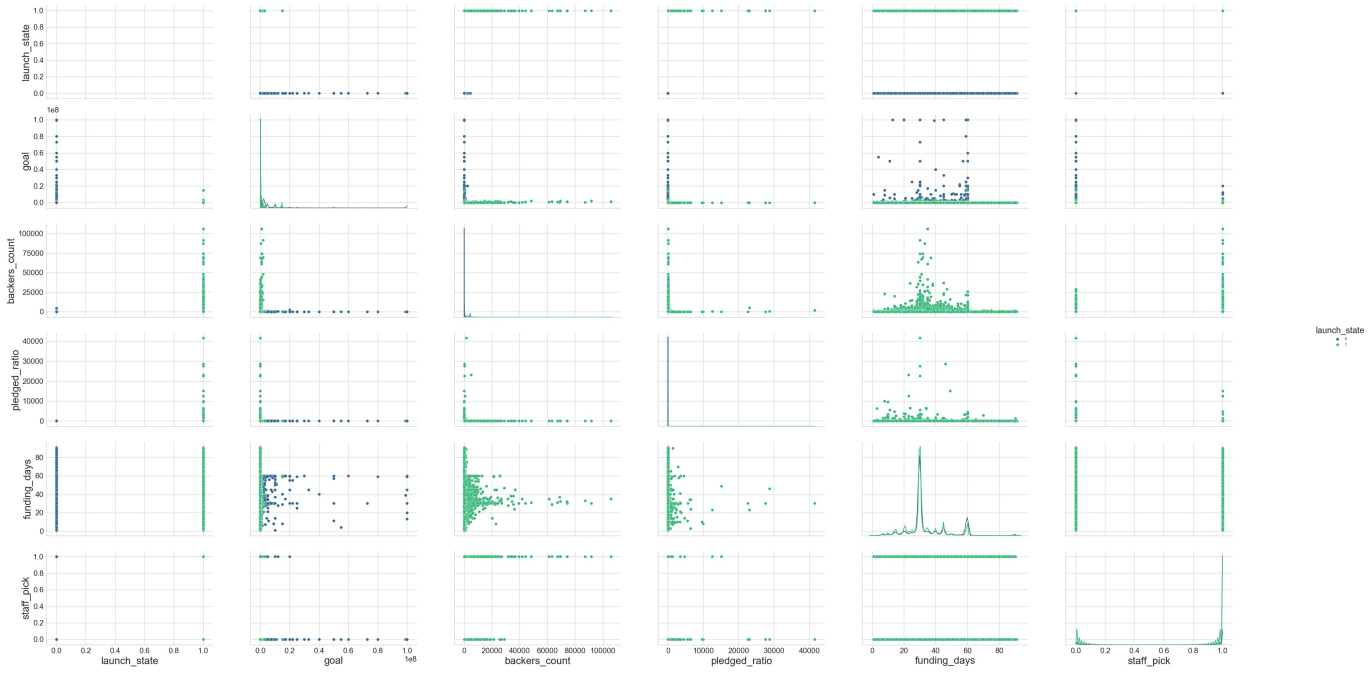
3.4 Exploratory data analysis

Refer to appendix A1.3 EDA for code.

It is always a good idea to do at least a bit of exploratory data analysis before diving into the machine learning aspect of a project. Creating visualizations can uncover interesting trends and also help guide further analysis.

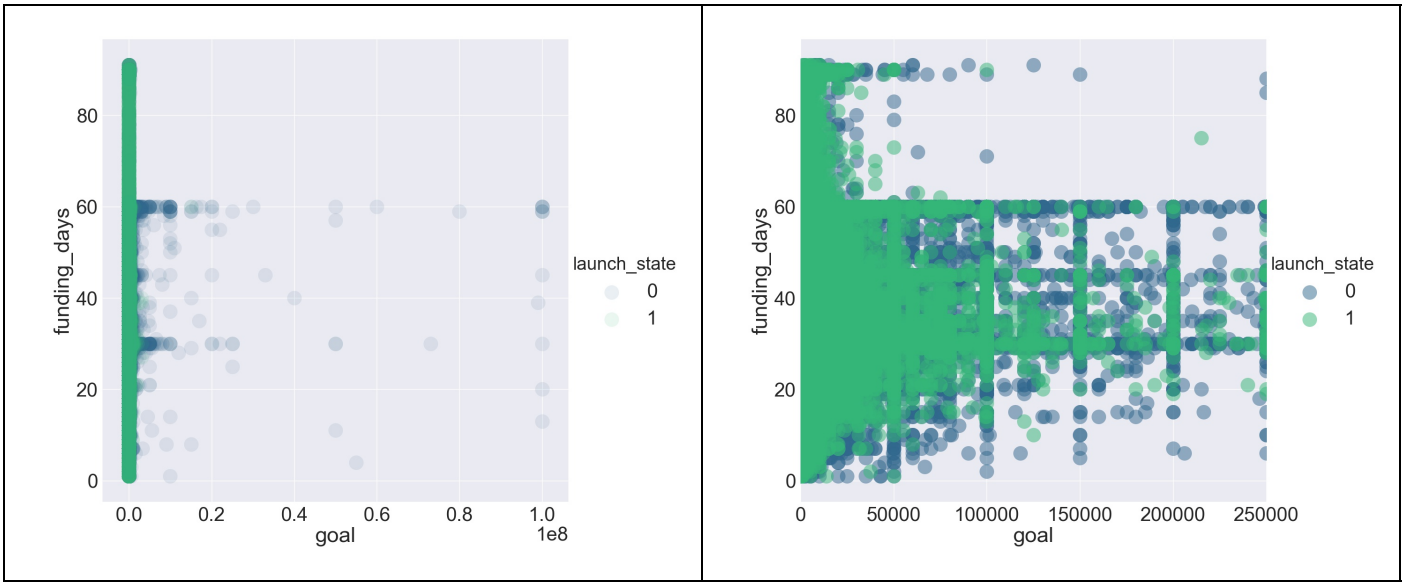
3.4.1 Pairplot

A pairplot of all of the non-dummy variables is shown below; successful projects are green while failed are blue. There does not seem to be good separation of *launch_state* values with the exception perhaps of *goal*: it does appear as if projects with large goals have few successes.



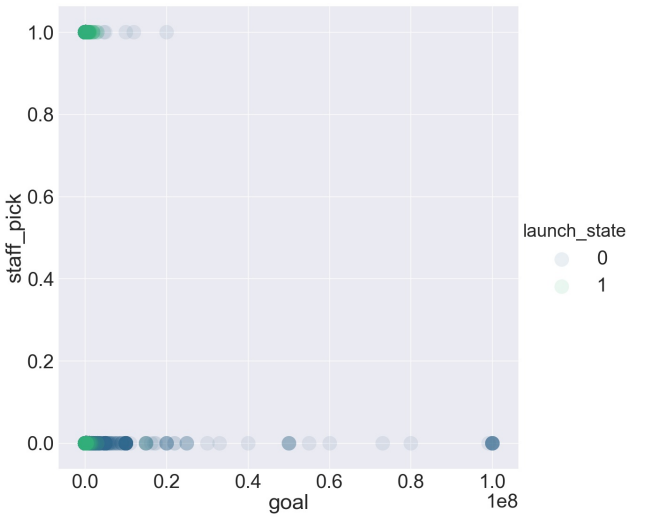
3.4.2 *funding_days* vs *goal*

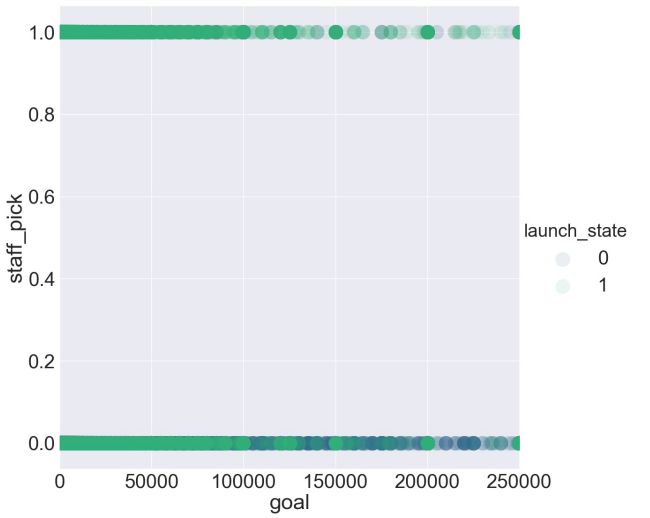
Below shows *funding_days* vs *goal*; the left plot shows all data points while the right is zoomed in on the *goal* (x-) axis to $0-250{,}000$. We can see that while there is no significant separation at these lower goal levels, there might be a slight benefit to having longer *funding_days*. Successful launches seem loosely clustered around *funding_days* = [0,60] and *goal* < $100k.



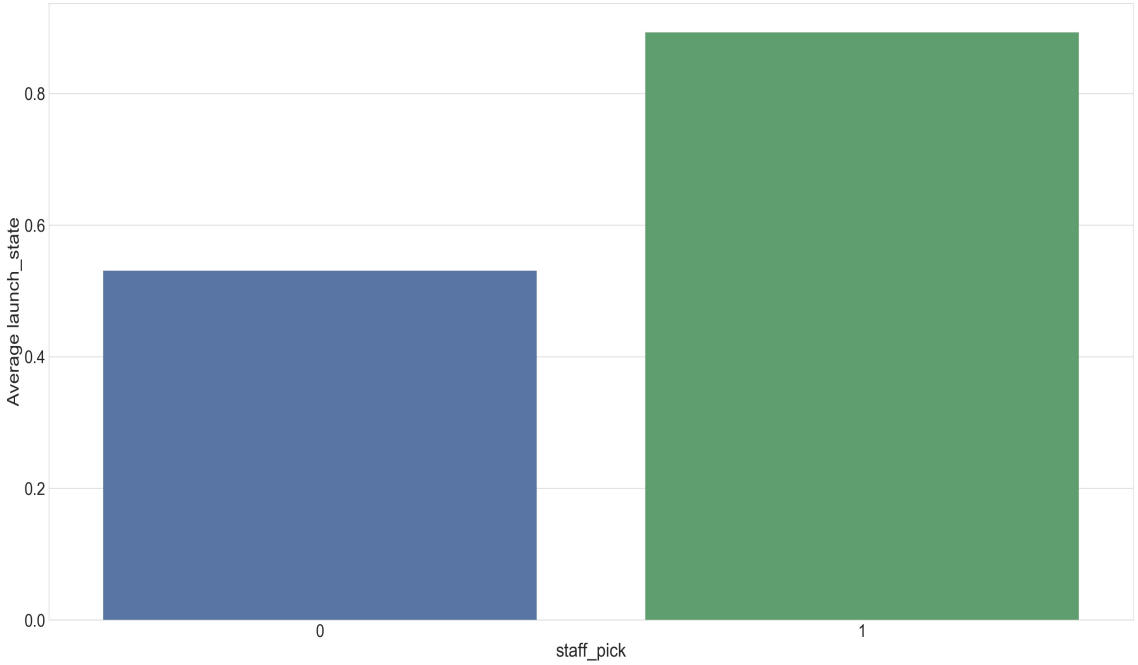
3.4.3 *staff_pick*

As mentioned above in the Variable - outcome correlation section, *staff_pick* has a relatively high correlation with the outcome of 0.25. Indeed, the images below show that projects chosen as a staff pick (*staff_pick* = 1) rarely fail to become successfully funded.





The bar plot below shows that the average *launch_state* for *staff_pick* of 0 and 1 is 0.523 and 0.889, respectively. Since *launch_state* is binary with 0 for failures and 1 for successes, these means also represent successful launch percentages, ie 52.3% of projects that are not chosen as a staff pick succeed while 88.9% of projects that are chosen succeed.

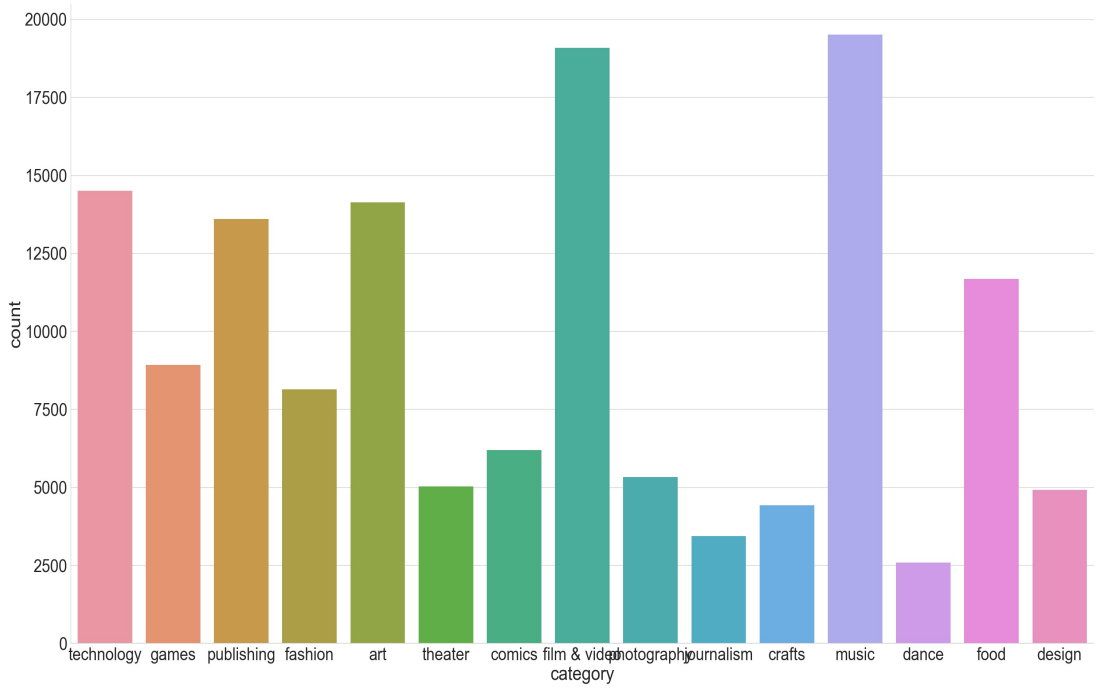


It should also be noted that 13.5% of the projects were chosen as staff picks. From https://www.kickstarter.com/blog/how-to-get-featured-on-kickstarter (https://www.kickstarter.com/blog/how-to-get-featured-on-kickstarter), it appears as if projects are featured when they catch the eye of the Kickstarter staff via creativity, a nice and visually appealing site, etc. ie, they are NOT just picked due to them being funded well.

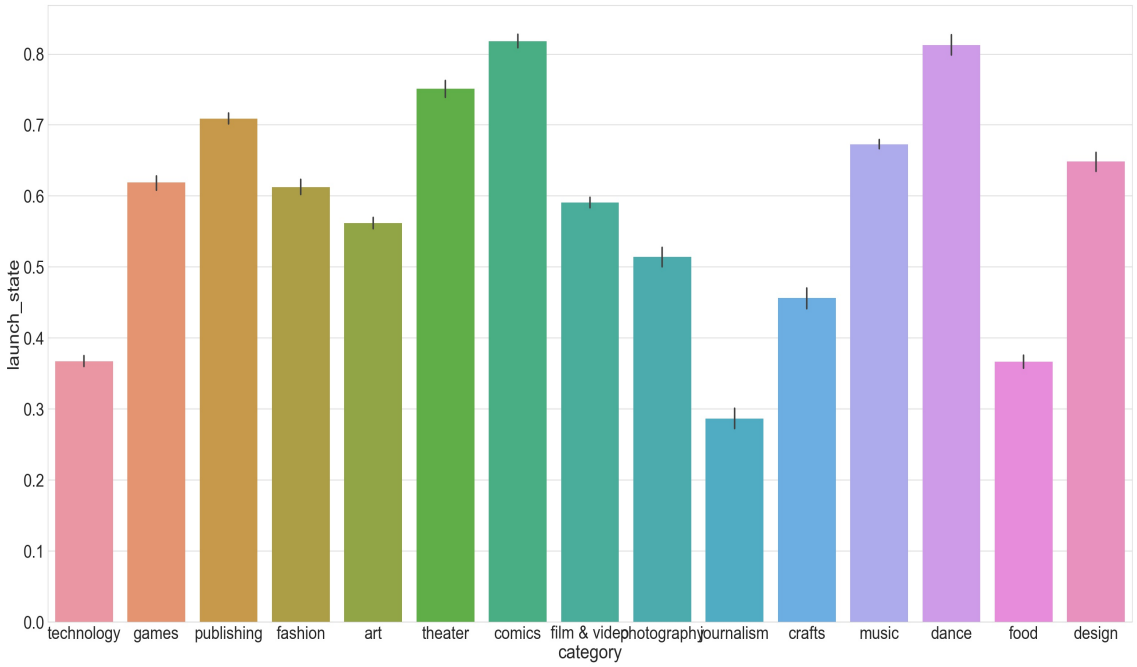
Clearly, being chosen as a staff pick is correlated with funding success. What is unclear, however, is whether getting chosen actually helps in success or if projects that were already going to be successful are chosen.

3.4.4 Categories

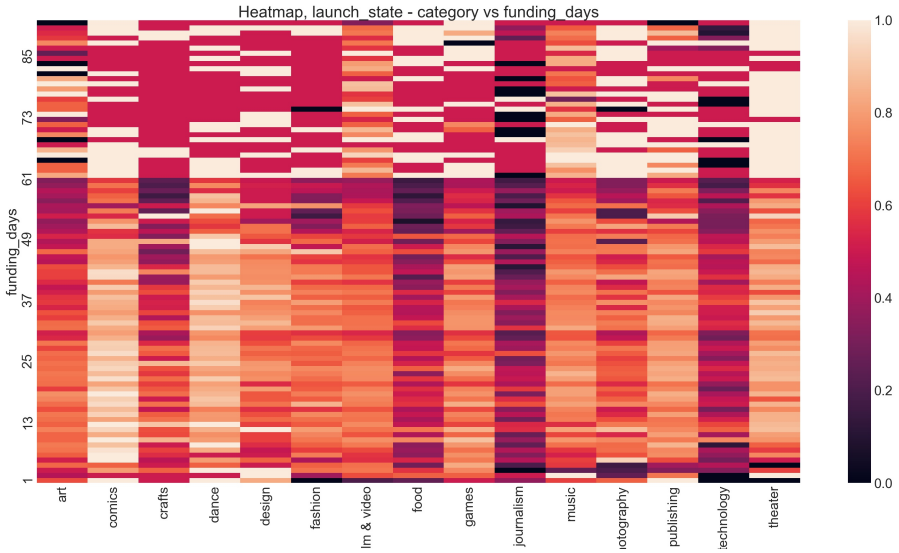
There are 15 main categories that a project can fall under (with hundreds of sub-categories). A frequency plot of these 15 categories is shown below. The two most common categories are 'film & video' and 'music'.

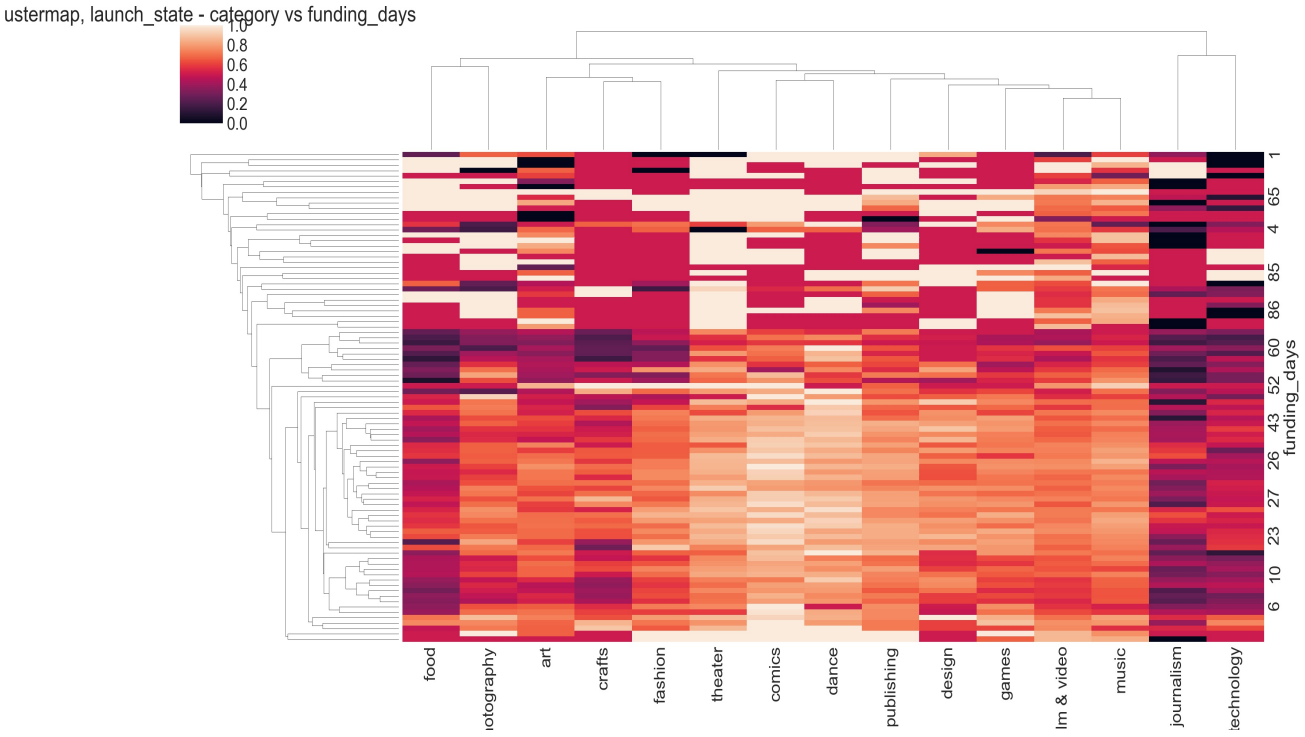


Note that the above plot says nothing of the success of these categories. For that, I've plotted the average *launch_state* value (which, again, can be interpreted as the percentage of successes) as a function of category, below. The three most successful categories are 'comics', 'dance', and 'publishing' while the least successful are 'journalism', 'technology', and 'food'.



To better show that some categories have better chances of success than other, I've plotted a heatmap and a clustermap, below. Successes (*launch_state* = 1) are white and failures (*luanch_state* = 0) are black. I used 0.5 for empty cells so as not to bias towards success or failure since a project can only be one category and so most values will indeed be empty. As such, in both figures, a mostly-white column can be interpreted as a highly successful category while a mostly-black column can be interpreted as not very successful. It can be clearly seen that, as indicated above, 'journalism' and 'technology' categories have a relatively high failure rate.





3.5 Model exploration

Refer to appendix A1.4 Model exploration for code.

To start creating the machine learning models, I separated the working dataset into train and test sets. Note that using 25% of the working set as the test size ensures that 20% of the entire raw set is reserved for the test set (this is not exact since we did some cleaning after creating the validation set, but it's close). Also note that we dropped the outcome *launch_state* variable as well as several information-only variables from the dataset to be used for machine learning variables, X.

```
info_variables = ['id','launched_at','category','country', 'pledged_ratio', 'backer
s_count']

X = df.drop(columns=info_variables).drop(columns='launch_state')
y = df['launch_state']

#----------------------------------------
# TRAIN/TEST SPLIT

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=101)
```

Next, the training and test datasets were scaled.

```
sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
```

With the data scaled, I built and analyzed all of the models of interest by

1. loading the relevant classifier.
2. fitting the training data and outcome vector to the classifier.
3. predicting the test outcomes using the trained classifier on the test data.
4. printing the confusion matrix and the classification report comparing the known test outcomes to the predicted outcomes.
5. calculating the one-run accuracy as the sum of the confusion matrix diagonal divided by the sum of the entire confusion matrix.
6. completing a 10-fold cross validation for models that did not take too long to run.
7. printing the 10-fold cross validation accuracies, mean accuracy, and accuracy standard deviation.

Again, all code can be found in A1.4 Model exploration.

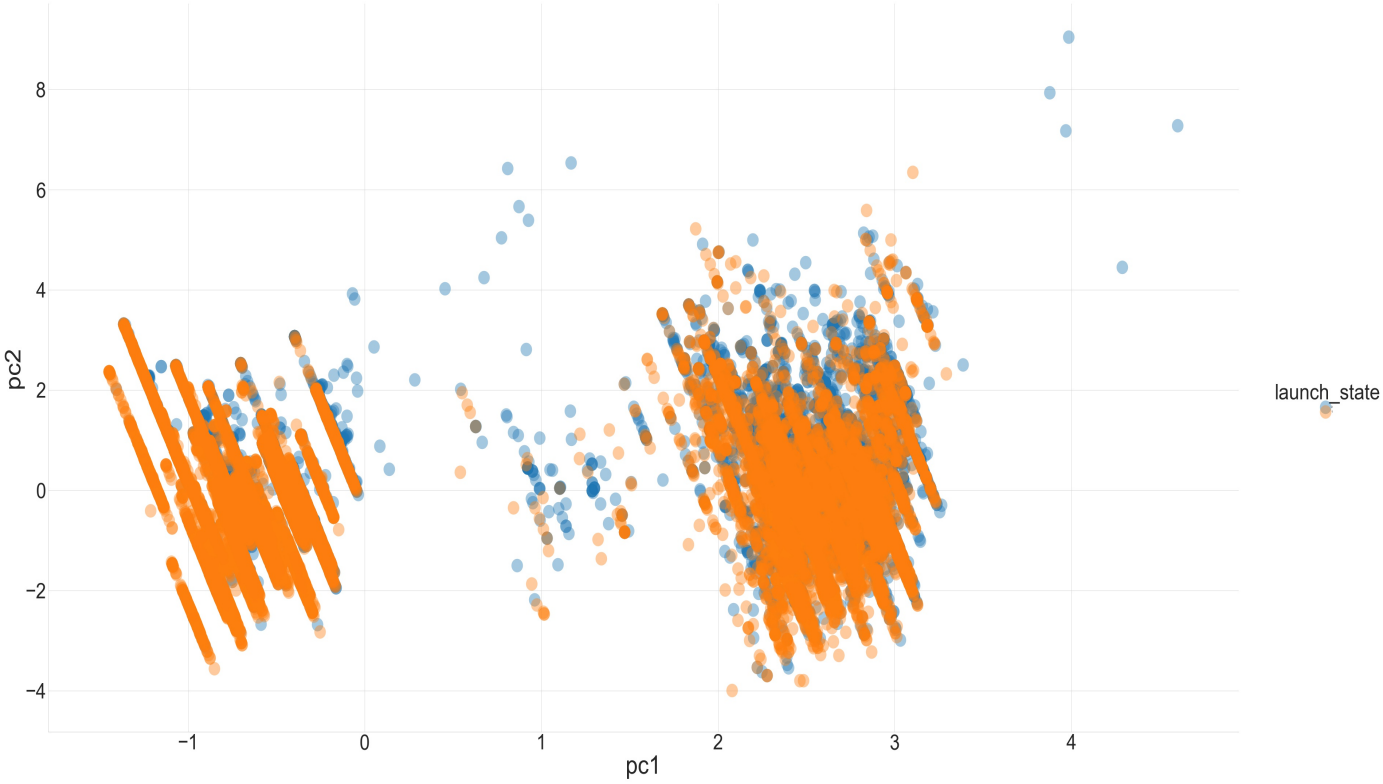
The results of these initial exploratory runs are shown in the table below.

```
df_results[0:7]
```

	model	time_fit	time_predict	time_10_fold_CV	accuracy	acc_10_fold
0	Naive Bayes	0.101767	0.036270	1.53579	0.623551	0.624725
1	Logistic Regression	1.822522	0.009085	17.0696	0.694333	0.689984
2	K Nearest Neighbors	7.340739	101.319590	344.234	0.682739	0.675807
3	SVM, Linear	954.375944	90.726687	None	0.677168	None
4	SVM, RBF	1091.777649	120.427602	None	0.682908	None
5	Decision Tree	0.437127	0.014542	4.82765	0.664357	0.662544
6	Random Forest (10-fold)	1.016743	0.104582	10.463	0.683248	0.680784

As can be seen above, all of the models seem to have similar accuracies (between ~62% and ~69%) while their run times vary greatly.

A quick look into variable reduction via principal component analysis (PCA) was then completed. The code for this is found in Appendix A1.4.9 Principal component analysis. Fitting the training data to a PCA object with only the top two principal components results in the following scatter plot. As expected (since there was not great separation when all of the variables were included), it does not seem like reducing the variables to their principal components make sense.



For completeness, the PCA object with two principal components was fit to the Naive Bayes algorithm. It ran very fast (about 5x faster than the original Naive Bayes model) and head nearly the same accuracy (61% instead of 62%). This would be a great approach in some cases to speed things up while maintaining similar accuracy! The results table below now includes this new PCA result.

```
df_results[0:8]
```

	model	time_fit	time_predict	time_10_fold_CV	accuracy	acc_10_fold
0	Naive Bayes	0.101767	0.036270	1.53579	0.623551	0.624725
1	Logistic Regression	1.822522	0.009085	17.0696	0.694333	0.689984
2	K Nearest Neighbors	7.340739	101.319590	344.234	0.682739	0.675807
3	SVM, Linear	954.375944	90.726687	None	0.677168	None
4	SVM, RBF	1091.777649	120.427602	None	0.682908	None
5	Decision Tree	0.437127	0.014542	4.82765	0.664357	0.662544
6	Random Forest (10-fold)	1.016743	0.104582	10.463	0.683248	0.680784
7	PCA (n=2), Naive Bayes	0.024343	0.003595	0.284533	0.609355	0.608927

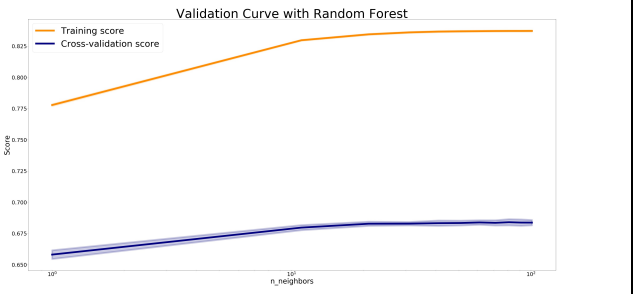
3.6 Model tuning and selection

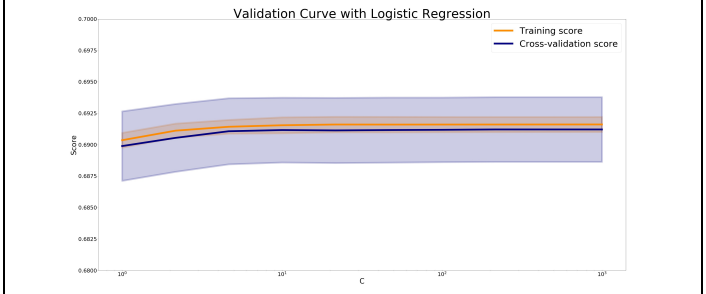
Refer to appendix A1.5 Model tuning and selection for code.

At this point it was time to choose some models to tune and make a final selection. This was completed for three models using sklearn's *GridSearchCV* method to hone in on the parameters that optimize the results. These optimal parameters are outlined below:

- Random forest:
  - *n_estimators* = 100
  - *criterion* = "gini"
  - *max_features* = "sqrt"
  - *min_samples_leaf* = 25
- Logistic regression:
  - *C* = 10
  - *penalty* = "l1"
- K nearest neighbors:
  - *metric* = "minkowski"
  - *n_neighbors* = 23
  - *p* = 2

We also plotted validation curves for the numeric parameters to double-check for over-fitting. These curves are shown below (**Note: The image of the KNN validation curve was unortunately not saved before the entire session was pickled and dumped, ie I do not have that particular plot**). Note that they agree with the above outline; the models are made complicated enough to minimize variance but not so complicated that bias becomes unnecessarily large. Interestingly, the training and cross-validation curves do not start deviating from each other (high bias) when the random forest mode's *n_neighbors* is high which is what I would expect from an over-fitted model.





The three models with optimized parameters were then run and results appended to the results table as shown below.

```
df_results
```

	model	time_fit	time_predict	time_10_fold_CV	accuracy	acc_10_fold
0	Naive Bayes	0.101767	0.036270	1.53579	0.623551	0.624725
1	Logistic Regression	1.822522	0.009085	17.0696	0.694333	0.689984
2	K Nearest Neighbors	7.340739	101.319590	344.234	0.682739	0.675807
3	SVM, Linear	954.375944	90.726687	None	0.677168	None
4	SVM, RBF	1091.777649	120.427602	None	0.682908	None
5	Decision Tree	0.437127	0.014542	4.82765	0.664357	0.662544
6	Random Forest (10-fold)	1.016743	0.104582	10.463	0.683248	0.680784
7	PCA (n=2), Naive Bayes	0.024343	0.003595	0.284533	0.609355	0.608927
8	Random Forest (Optimized)	5.856856	0.470576	66.9746	0.718257	0.716067
9	Logistic Regression (Optimized)	6.454474	0.003956	68.8582	0.695662	0.691059
10	KNN (Optimized)	7.253727	48.397783	211.557	0.700328	0.69877

As shown above, the prediction accuracy increases slightly for all three models when using optimized parameters. An optimized random forest model provides the best accuracy at ~72% while at the same time remaining efficient and fast to run. This is the chosen model for future use.

4 Prediction / final validation

Refer to appendix A2 Code - new data prediction for relevant code.

With one specific model chosen to launch, it is time for a final validation. In this case, this validation also serves as a proving grounds for the final product. Recall that thus far we have not touched 20% of the initial raw data - this untouched set was set aside as a validation set. Keeping a completely separated validation set is useful because it ensures that the model in no way relied on it. This is in contrast to the training set which was used extensively to fit and train the model. Even the test set was used indirectly to baseline the model when doing accuracy comparisons. Only the validation set has been 100% unused.

The goal here is to then create a script that takes in raw Kickstarter project data (again, in this case that raw data is the validation set previously set aside), cleans it up, and uses the chosen machine learning algorithm to predict which projects will be successfully funded or not. Since we do have the real-life outcomes of this set, we can also calculate our final product accuracy.

Aside from some extra code that asks the user for input, runs various handling exceptions, and deals with whether or not the input file is a truly raw JSON file or whether it is an extracted dataframe from the JSON file (as is the case here with the validation set), the product script is quite simple. It

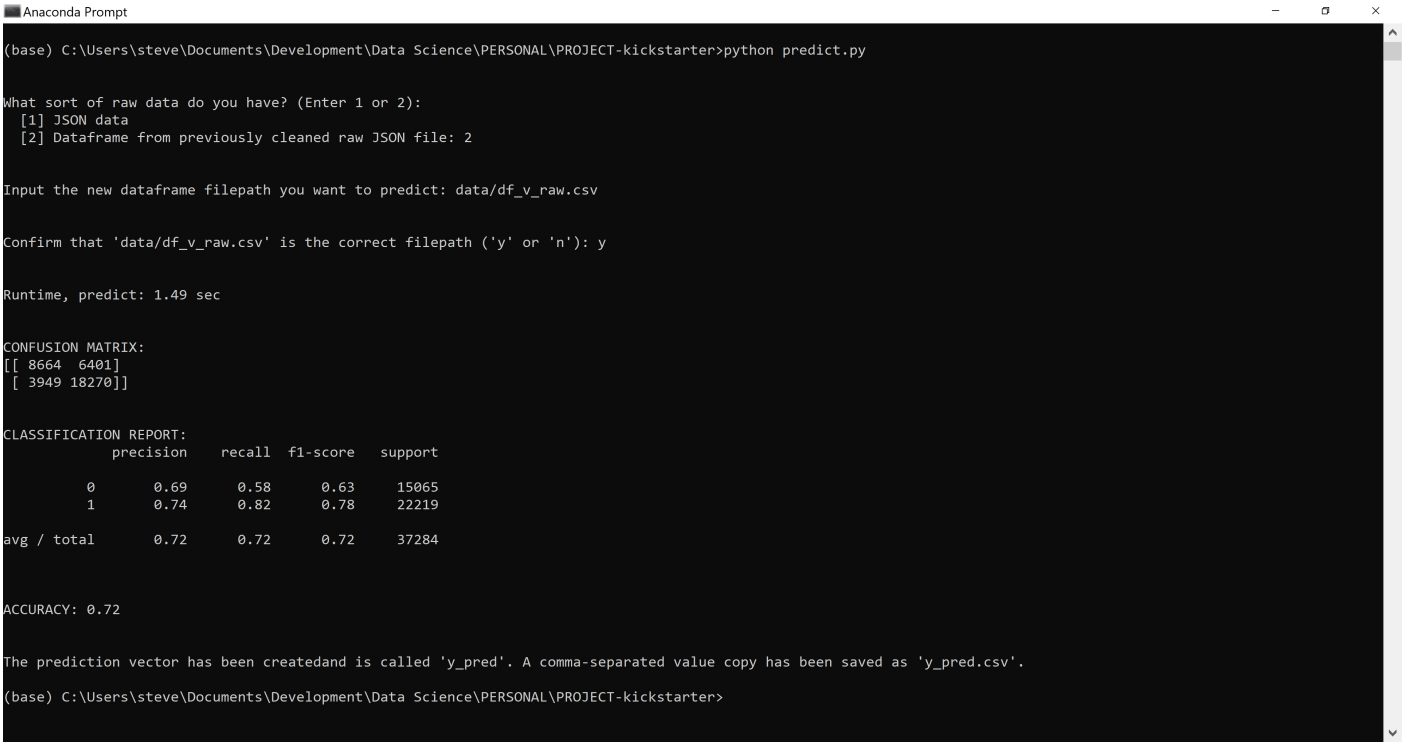
1. takes in the raw data.
2. cleans the data.
3. extracts the relevant machine learning columns as well as the outcome column (if one exists).
4. applies the prediction algorithm.
5. writes out the prediction vector to a CSV file.
6. evaluates the accuracy of the prediction vector to the known values, if applicable. Specifically, it prints the confusion matrix, classification report, and calculates the accuracy as the sum of the confusion matrix diagonal divided by the sum of the entire confusion matrix.

4.1 Instructions

Running the script is quite simple.

1. Ensure that Python is properly installed.
2. Ensure that the following files are located in the working directly:
    - classifier_rf_opt.pkl
    - f_cleanData.py
    - f_dataImport.py
    - f_predict.py
    - predict.py
    - sc_X.pkl
3. Create a folder named 'data' in the working directory.
4. Download or create the Kickstarter data to run the prediction model on.
    - The data must be in JSON format like from https://webrobots.io/kickstarter-datasets/ (https://webrobots.io/kickstarter-datasets/).
    - Alternatively, the data can a comma-separated value dataframe from previously-run analyses.
5. Save the raw data in the 'data' folder.
6. Open a command prompt.
7. Type 'python predict.py'
8. Follow the prompts.

A screen shot of this process is shown below.



The above image also shows the confusion matrix, classification report, and calculated accuracy. As expected, **the model performs admirably with a 70% accuracy and managed to complete the analysis in only 1.49 seconds**. Further, a copy of the prediction vector was saved in the home folder.

5 Next steps

Some recommendations to improve this analysis include:

- Complete an analysis to determine the statistical influence of the 'staff_pick' variable, ie while it appears as if staff_pick = 1 results in a higher chance that a project is successfully funded, is this statistically accurate?
- Re-run the analysis without the country dummy variables.
- Re-run the analysis without the category dummy variables.
- Code the finished script so that it can accept raw data without every column. Specifically, a small enough amount of raw data may not include all of the required countries or categories currently required to fit the model.
- Analyze the effect of including sub-categories in the analysis.
- Look for trends in the failed predictions.

Appendix

A1 Code - model creation

A1.1 Data preparation function (as of 2018-12-12)

Relevant file: 'f_dataImport.py'

```
# -*- coding: utf-8 -*-
"""
Created on Tue Nov 27 14:06:31 2018

@author: steve
"""



#==============================================================================
#
# IMPORT LIBRARIES
#
#==============================================================================
import pandas as pd
import os
from pandas.io.json import json_normalize
import json

#==============================================================================
#
# FUNCTION - RAW DATA IMPORT
#
#==============================================================================

def dataImport():
    '''
    This function imports the raw json data downloaded from
    https://webrobots.io/kickstarter-datasets/ and extracts the dictionaries
    into a usable dataframe format.

    OUTPUTS:
        * 'df_raw.csv': raw data dataframe
    '''


    #----------------------------------------
    # READ IN RAW DATA

    print('\n')
    print('***')
    print('NOTE:')
    print('The input file must be JSON with the same format as those'
          'downloaded from https://webrobots.io/kickstarter-datasets/')
    print('***')

    while True:

        print('\n')
```

```
        new_data = str(input('Input the new data JSON filepath you want to predict:
'))

        if not os.path.exists(new_data):
            print('\n')
            print('The filepath \'', new_data, '\' does not exist.', sep='')
            continue
        else:
            print('\n')
            yesno = str(input(f'Confirm that \'{new_data}\' is the correct '
                              'filepath (\'y\' or \'n\'): '))

            if (yesno[0].lower() == "y"):
                with open(new_data, encoding="utf8") as json_file:
                     json_obj = [json.loads(line) for line in json_file]
                break
            elif (yesno[0].lower() == 'n'):
                print('\n')
                continue
            else:
                print('\n')
                print('Improper input')
                continue

    #----------------------------------------
    # UNPACK RAW DATA

    json_obj2 = []
    # append 'data' dictionary only
    for x in range(0, len(json_obj)):
        json_obj2.append(json_obj[x]["data"])

    # Check
    if (len(json_obj2) - len(json_obj)) != 0:
        print('*** ERROR: Did not extract all json \'data\' entries ***')

    # ---- CONVERT TO DATAFRAME ----
    df_raw_json = pd.DataFrame(json_obj2)

    # ---- UNPACK DICTIONARY ENTRIES ----
    # 'category'
    df_category = json_normalize(data=df_raw_json['category'])
    df_category.columns = 'category_' + df_category.columns

    # 'creator'
    df_creator = json_normalize(data=df_raw_json['creator'])
    df_creator.columns = 'creator_' + df_creator.columns

    # 'location'
```

```
    # Must mannually unpack 'location' with pd.Series due to NaN elements
    df_location = df_raw_json['location'].apply(pd.Series)
    df_location.drop(columns=0, inplace=True)
    df_location.columns = 'location_'+df_location.columns
    df_location1 = df_location['location_urls'].apply(pd.Series)
    df_location1.drop(columns=0, inplace=True)
    df_location1.columns = 'location_urls_'+df_location1.columns
    df_location2 = df_location1['location_urls_web'].apply(pd.Series)
    df_location2.drop(columns=0, inplace=True)
    df_location2.columns = 'location_urls_web_'+df_location2.columns
    df_location3 = df_location1['location_urls_api'].apply(pd.Series)
    df_location3.drop(columns=0, inplace=True)
    df_location3.columns = 'location_urls_api_'+df_location3.columns
    # Concat 'location' dataframes
    df_location = pd.concat([df_location, df_location2, df_location3], axis=1)
    df_location.drop(columns='location_urls', inplace=True)

    # 'photo'
    df_photo = json_normalize(data=df_raw_json['photo'])
    df_photo.columns = 'photo_' + df_photo.columns

    # 'profile'
    df_profile = json_normalize(data=df_raw_json['profile'])
    df_profile.columns = 'profile_' + df_profile.columns

    # 'urls'
    df_urls = json_normalize(data=df_raw_json['urls'])
    df_urls.columns = 'urls_' + df_urls.columns

    # ---- CONCAT UNPACKED DATAFRAMES ----
    df_raw = pd.concat([df_raw_json, df_category, df_creator, df_location,
                        df_photo, df_profile, df_urls], axis=1)
    df_raw.drop(columns=['category','creator','location','photo',
                         'profile','urls'], inplace=True)

    #----------------------------------------
    # WRITE OUT

    df_raw.to_csv('data/df_raw.csv', sep=",")

    return df_raw
```

A1.2 Data cleaning function (as of 2018-12-12)

Relevant file: 'f_cleanData.py'

```
# -*- coding: utf-8 -*-
"""
Created on Mon Dec 10 13:04:33 2018

@author: steve
"""

#==============================================================================
#
# IMPORT LIBRARIES
#
#==============================================================================

import pandas as pd
from datetime import datetime

#==============================================================================
#
# FUNCTION - CLEAN DATA
#
#==============================================================================
def cleanData(df):
    '''
    This function cleans downloaded raw data for the Kickstarter success
    prediction project. The input dataframe must be imported and the json
    extracted using '00 - Data Import.py'. There should be either 95 or 96
    columns (the 'state' column is optional) and they must be labeled exactly
    as defined in '00 - Data Import.py'.
    '''

    #----------------------------------------
    # ADD EMPTY STATE COLUMN IF NECESSARY
    if 'state' not in df.columns:
        df['state'] = None

    #----------------------------------------
    # COLUMN CLEANUP

    drop_vars = ['photo_1024x576', 'photo_1536x864', 'photo_ed', 'photo_full',
                 'photo_key', 'photo_little', 'photo_med', 'photo_small',
                 'photo_thumb', 'slug', 'urls_api.message_creator', 'urls_api.sta
r',
                 'urls_web.message_creator', 'urls_web.project', 'urls_web.reward
s',
                 'source_url', 'creator_avatar.medium', 'creator_avatar.small',
                 'creator_avatar.thumb', 'creator_chosen_currency', 'creator_id',
                 'creator_name', 'creator_slug', 'creator_urls.api.user',
```

```
                'creator_urls.web.user', 'location_id', 'location_name',
                'location_slug', 'location_short_name', 'location_displayable_nam
e',
                'location_localized_name', 'location_type', 'location_is_root',
                'location_urls_web_discover', 'location_urls_web_location',
                'location_urls_api_nearby_projects', 'category_color',
                'category_id', 'category_urls.web.discover',
                'profile_background_color',
                'profile_background_image_attributes.id',
                'profile_background_image_attributes.image_urls.baseball_card',
                'profile_background_image_attributes.image_urls.default',
                'profile_background_image_opacity', 'profile_blurb',
                'profile_feature_image_attributes.id',
                'profile_feature_image_attributes.image_urls.baseball_card',
                'profile_feature_image_attributes.image_urls.default', 'profile_i
d',
                'profile_link_background_color', 'profile_link_text',
                'profile_link_text_color', 'profile_link_url', 'profile_name',
                'profile_project_id', 'profile_should_show_feature_image_section',

                'profile_show_feature_image', 'profile_state',
                'profile_state_changed_at', 'profile_text_color', 'currency_symbo
l',
                'static_usd_rate','converted_pledged_amount','fx_rate',
                'current_currency', 'usd_pledged', 'is_starrable', 'friends',
                'is_backing', 'is_starred', 'permissions', 'name', 'blurb',
                'location_state', 'location_country', 'currency',
                'currency_trailing_code', 'state_changed_at', 'category_parent_i
d',
                'category_position', 'category_name', 'category_id',
                'creator_is_registered', 'disable_communication', 'created_at',
                'usd_type']

    df.drop(columns=drop_vars, inplace=True)

    # Rename columns
    df.rename(columns={'category_slug':'category', 'state':'launch_state'}, inplace
=True)

    # Rearrange columns
    df = df[['launch_state', 'id', 'category', 'goal', 'backers_count',
             'pledged', 'country','deadline', 'launched_at',
             'staff_pick', 'spotlight']]

    #----------------------------------------
    # EXTRACT CATEGORIES
    df['category'] = [i.split('/')[0] for i in df['category']]

    #----------------------------------------
```

```
    # REMOVE DUPLICATES
    df.drop_duplicates(inplace=True)
    # For duplicate IDs leftover, remove the lesser pledged row
    df = df.sort_values('pledged', ascending=False).drop_duplicates('id').sort_inde
x()

    # Check
    if (len(df) - len(df["id"])) != 0:
        print('*** WARNING: There are ',
              len(df) - len(df["id"]),
              ' duplicate IDs ***', sep='')

    #----------------------------------------
    # CONVERT DATETIMES
    df['deadline'] = df['deadline'].apply(datetime.utcfromtimestamp)
    df['launched_at'] = df['launched_at'].apply(datetime.utcfromtimestamp)

    #----------------------------------------
    # NA IMPUTATION
    # Checks
    if df.isnull().sum().sum() != 0:
        print('*** WARNING: There are null values ***')
    if df.isna().sum().sum() != 0:
        print('*** WARNING: There are NA values ***')
    if (df=='').sum().sum() != 0:
        print('*** WARNING: There are empty string (\'\') values ***')

    #----------------------------------------
    # CLEAN UP 'launch_state'
    df.query("launch_state == 'failed' | "
             "launch_state == 'successful' | "
             "launch_state == None", inplace=True)

    #----------------------------------------
    # CONVERT CATEGORICAL VARIABLES TO DUMMY VARIABLES
    category = pd.get_dummies(df['category'], drop_first=True)
    country = pd.get_dummies(df['country'], drop_first=True)
    d_launch_state = dict(zip(['failed','successful'], range(0,2)))
    launch_state = df['launch_state'].map(d_launch_state)

    # Check
    if (df[df['launch_state'] == 'successful'].shape[0] - launch_state.sum() != 0):
        print('*** WARNING: Some launch_states did not map to 0/1 ***')

    # Drop the categorical launch_state column
    # (keep 'category' and 'country' for  visualization)
    df.drop(['launch_state'],axis=1,inplace=True)

    # Add the new dummy variable launch_state column and move it to column index 0
```

```
    # and country to column index 3
    df = pd.concat([launch_state, df], axis=1)
    df = df[['launch_state', 'id', 'category', 'country', 'goal', 'backers_count',
             'pledged','deadline', 'launched_at', 'staff_pick', 'spotlight']]

    # Add the dummy variable country and category columns
    df = pd.concat([df, category, country], axis=1)

    # Checks
    if (df.isnull().sum().sum() != 0):
        print('*** WARNING: Null values introduced with dummy variables ***')
    if (df.isna().sum().sum() != 0):
        print('*** WARNING: NA values introduced with dummy variables ***')
    if (df=='').sum().sum() != 0:
        print('*** WARNING: Empty string (\'\') values introduced with dummy variab
les ***')

    #----------------------------------------
    # FINAL CLEANUP
    # pledged_ratio
    pledged_ratio = df['pledged'] / df['goal']
    df.insert(loc=df.columns.get_loc("pledged"), column='pledged_ratio',
              value=pledged_ratio)
    df.drop(columns='pledged', inplace=True)

    # datetime columns
    funding_days = (df['deadline'] - df['launched_at']).dt.days
    df.insert(loc=df.columns.get_loc("deadline"), column='funding_days',
              value=funding_days)
    df.drop(columns='deadline', inplace=True)

    # ---- MOVE 'LAUNCHED_AT' ----
    launched_at = df['launched_at']
    df.drop(columns='launched_at', inplace=True)
    df.insert(loc=2, column='launched_at', value=launched_at)

    # ---- CONVERT 'STAFF_PICK' AND 'SPOTLIGHT' TO DUMMIES ----
    d_staff_pick = dict(zip([False,True], range(0,2)))
    staff_pick = df['staff_pick'].map(d_staff_pick)

    # Check
    if (df[df['staff_pick'] == True].shape[0] - staff_pick.sum()) != 0:
        print('*** WARNING: \'staff_pick\' not mapped to 0/1 properly ***')

    d_spotlight = dict(zip([False,True], range(0,2)))
    spotlight = df['spotlight'].map(d_spotlight)

    # Check
    if (df[df['spotlight'] == True].shape[0] - spotlight.sum()) != 0:
```

```
        print('*** WARNING: \'spotlight\' not mapped to 0/1 properly ***')

    df.drop(['staff_pick','spotlight'],axis=1,inplace=True)

    df.insert(loc=df.columns.get_loc("comics"), column='staff_pick', value=staff_pi
ck)
    df.insert(loc=df.columns.get_loc("comics"), column='spotlight', value=spotligh
t)

    #----------------------------------------
    # VARIABLE REDUCTION
    df.drop(columns='spotlight', inplace=True)

    # ---- NULL/NA/EMPTY CHECKS ----
    if (df.isnull().sum().sum() != 0):
        print('*** WARNING: Null values introduced with \'staff_pick\' and \'spotli
ght\' dummy variables ***')
    if (df.isna().sum().sum() != 0):
        print('*** WARNING: NA values introduced with \'staff_pick\' and \'spotligh
t\' dummy variables ***')
    if (df=='').sum().sum() != 0:
        print('*** WARNING: Empty string (\'\') values introduced with \'staff_pick
\' and \'spotlight\' dummy variables ***')

    #----------------------------------------
    # WRITE OUT
    df.to_csv('df_clean.csv', sep=",")

    return df
```

A1.3 EDA (as of 2018-12-12)

Relevant file: '03 - EDA.py'

```
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 15 14:43:36 2018

@author: steve
"""

#----------------------------------------
# USER INPUTS



#----------------------------------------
# IMPORT LIBRARIES

import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

#----------------------------------------
# IMPORT DATAFRAME

df = pd.read_csv('data/df02.csv', sep=',', na_filter=False, index_col=0,
                 parse_dates=['launched_at'])

# Checks
if (df.isnull().sum().sum() != 0):
    print('*** WARNING: Null values introduced with read_csv ***')
if (df.isna().sum().sum() != 0):
    print('*** WARNING: NA values introduced with read_csv ***')
if (df=='').sum().sum() != 0:
    print('*** WARNING: Empty string (\'\') values introduced with read_csv ***')

#----------------------------------------
# EXPLORATORY DATA ANALYSIS

# ---- EXPLORE launch_state ----
fig=plt.figure(figsize=(8,4))
sns.set_style('whitegrid')
sns.countplot(x='launch_state', data=df, palette='viridis')

# Success rate is proving difficult to calculate due to dirty (especially
# duplicate data. TBD)

# ============================================================================
# It looks like a huge portion of projects ultimately fail to launch! We will
# look into this more later, but for now consider the following from
# https://www.kickstarter.com/help/stats (as of 2018-11-16 10:40):
```

```
#  * Overall success rate: 36.53%
#      - Why is this different than the 53.2% success rate calculted above?
#  * "While 13% of projects finished having never received a single pledge 78%
#    of projects that raised more than 20% of their goal were successfully
#    funded."
# ============================================================================

# ---- DUMMY VARIABLE PAIRPLOT ----
sns.set_context("paper", rc={"axes.labelsize":20,
                             "xtick.labelsize": 15, "ytick.labelsize": 15})
sns.set_style('whitegrid')
plt.figure(figsize=(14,5))
sns.pairplot(data=df.drop(df.columns[10:], axis=1).drop(
        columns=['id','launched_at','category','country']),
             diag_kind='kde', hue='launch_state', palette='viridis')
# Observation: little insight to be gained here

# ---- FUNDING_DAYS VS GOAL ----
sns.set_style('whitegrid')
sns.set(font_scale=3)
sns.lmplot("goal", "funding_days", data=df, hue='launch_state', size=12,
           fit_reg=False, scatter_kws={'alpha':0.1, 's':500}, palette='viridis')

sns.set_style('whitegrid')
sns.set(font_scale=3)
sns.lmplot("goal", "funding_days", data=df, hue='launch_state', size=12,palette='vi
ridis',
           fit_reg=False, scatter_kws={'alpha':0.5, 's':500}).set(xlim=(0,250000))

# ============================================================================
# Observations:
#  * Successful launches seem loosely clustered around funding_days = [0,60]
#    and goal < $100k
# ============================================================================

# ---- BACKERS_COUNT VS GOAL ----
sns.set_style('whitegrid')
sns.set(font_scale=3)
sns.lmplot("goal", "backers_count", data=df, hue='launch_state', size=12,
           fit_reg=False, scatter_kws={'alpha':0.1, 's':500}, palette='viridis')

sns.set_style('whitegrid')
sns.set(font_scale=3)
sns.lmplot("goal", "backers_count", data=df, hue='launch_state', size=12,
           fit_reg=False, scatter_kws={'alpha':0.1, 's':500}, palette='viridis').se
t(
    xlim=(0,250000), ylim=(0,15000))

sns.set_style('whitegrid')
```

```
sns.set(font_scale=3)
sns.lmplot("goal", "backers_count", data=df, hue='launch_state', size=12,
           fit_reg=False, scatter_kws={'alpha':0.1, 's':500}, palette='viridis').se
t(
    xlim=(0,100000), ylim=(0,2000))

# ================================================================================
# Observations: backers_count is not a good predictor; it's obvious that
# with a high number of backers the project is more likely to succeed. More
# important for this project, though, is the fact that it cannot be used a priori.
# ================================================================================

# ---- PLEDGED_RATIO vs GOAL ----
sns.set_style('whitegrid')
sns.set(font_scale=3)
sns.lmplot("goal", "pledged_ratio", data=df, hue='launch_state', size=12,
           fit_reg=False, scatter_kws={'alpha':0.1, 's':500}, palette='viridis')

sns.set_style('whitegrid')
sns.set(font_scale=3)
sns.lmplot("goal", "pledged_ratio", data=df, hue='launch_state', size=12,
           fit_reg=False, scatter_kws={'alpha':0.1, 's':500},
           palette='viridis').set(xlim=(0,250000), ylim=(0,2))

# ================================================================================
# Observations: As expected, pledged_ratio < 1 means failure and all
# pledged_ratio >= 1 means success
# ================================================================================

# ---- STAFF_PICK vs GOAL ----
sns.set_style('whitegrid')
sns.set(font_scale=3)
sns.lmplot("goal", "staff_pick", data=df, hue='launch_state', size=12,
           fit_reg=False, scatter_kws={'alpha':0.1, 's':500}, palette='viridis')

sns.set_style('whitegrid')
sns.set(font_scale=3)
sns.lmplot("goal", "staff_pick", data=df, hue='launch_state', size=12,
           fit_reg=False, scatter_kws={'alpha':0.1, 's':500},
           palette='viridis').set(xlim=(0,250000))

# Observations: Staff_pick seems to be a decent indicator in launch_state

# ---- EXPLORE STAFF_PICK AND LAUNCH_STATE
df_staff_picks = df[['launch_state','staff_pick']].groupby(
        ['staff_pick'], as_index=False).count()
df_staff_picks.columns = ['staff_pick','freq']
df_staff_picks['ratio'] = df[['launch_state','staff_pick']].groupby(
        ['staff_pick'], as_index=False).mean()['launch_state']
```

```
plt.figure()
sns.set_style('whitegrid')
ax = sns.barplot(data = df[['launch_state','staff_pick']].groupby(
        ['staff_pick'], as_index=False).mean(), x='staff_pick', y='launch_state')
ax.set(xlabel='staff_pick', ylabel='Average launch_state')

# ============================================================================
# Observations:
#   * 13.5% of projects are chosen as staff picks
#   * staff_pick seems to correlate with launch_state:
#     - 52.3% of projects not chosen as staff picks succeed
#     - 88.9% of projects chosen as staff picks succeed
#
# From https://www.kickstarter.com/blog/how-to-get-featured-on-kickstarter,
# it appears as if projects are featured when they catch the eye of the
# Kickstarter staff via creativity, a nice and visually appealing site, etc.
# ie, they are NOT just picked due to them being funded well.
# ============================================================================

# ============================================================================
# So what have we learned so far?
# * goal - use it
# * backers_count - do not use it (it's not known beforehand)
# * pledged_ratio - do not use it (it's just an indicator of success)
# * funding_days - use it
# * staff_pick - use it
# ============================================================================

# ---- VISUALIZE CATEGORIES ----
df_categories = df[['launch_state','category']].groupby(
        ["category"]).describe().reset_index()
df_categories.sort_values(by=[('launch_state','mean')], ascending=False)
print(df_categories)

# Frequency plot
sns.set_style('whitegrid')
sns.factorplot(x='category', data=df, kind='count', size=10)

# Success ratio plot
plt.figure()
sns.set_style('whitegrid')
sns.barplot(x='category',y='launch_state',data=df)

# Observations: clearly, some categories are more successful than others.

# Heatmap
# Note that I fill in empty cells with 0.5 so as not to bias towards
# 0 (failure) and 1 (success)
```

```
plt.figure()
sns.set_style('whitegrid')
ax = sns.heatmap(
        df.pivot_table(values='launch_state', columns='category',
                       index='funding_days', fill_value=0.5), xticklabels=True)
ax.invert_yaxis()
plt.title('Heatmap, launch_state - category vs funding_days')

# Clustermap
sns.set_style('whitegrid')
sns.clustermap(df.pivot_table(values='launch_state', columns='category',
                              index='funding_days', fill_value=0.5),
               xticklabels=True)
plt.title('Clustermap, launch_state - category vs funding_days')
```

A1.4 Model exploration (as of 2018-12-12)

Relevant file: '04 Working - Models.ipynb'

A1.4.1 Model setup

```
#----------------------------------------
# IMPORT LIBRARIES

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import time

pd.options.display.max_columns = None # Shows all columns

from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.model_selection import cross_val_score
from sklearn.model_selection import validation_curve

from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.decomposition import PCA
from sklearn.model_selection import GridSearchCV

#----------------------------------------
# DUMP/LOAD SESSIONS

import dill

# ---- FULL SESSIONS ----
#dill.dump_session('./04 Working - Models.db')
#dill.load_session('./04 Working - Models.db')

# ---- OBJECTS ----
#dill.dump(sc_X, open("sc_X.pkl", "wb"))
#dill.dump(classifier_rf_opt, open("classifier_rf_opt.pkl", "wb"))
#dill.dump(df_results, open("df_results.pkl", "wb"))

#----------------------------------------
# IMPORT DATAFRAME

df = pd.read_csv('data/df02.csv', sep=',', na_filter=False, index_col=0,
                 parse_dates=['launched_at'])

# Checks
```

```
if (df.isnull().sum().sum() != 0):
    print('*** WARNING: Null values introduced with read_csv ***')
if (df.isna().sum().sum() != 0):
    print('*** WARNING: NA values introduced with read_csv ***')
if (df=='').sum().sum() != 0:
    print('*** WARNING: Empty string (\'\') values introduced with read_csv ***')

info_variables = ['id','launched_at','category','country', 'pledged_ratio', 'backer
s_count']

X = df.drop(columns=info_variables).drop(columns='launch_state')
y = df['launch_state']

#----------------------------------------
# TRAIN/TEST SPLIT

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.25, random_state=101)

#----------------------------------------
# FEATURE SCALING

sc_X = StandardScaler()
X_train = sc_X.fit_transform(X_train)
X_test = sc_X.transform(X_test)
```

A1.4.2 Naive Bayes

```
#==============================================================================
#
# NAIVE BAYES
#
#==============================================================================

#----------------------------------------
# FIT MODEL

start_clock = time.clock()

# Naive-Bayes
classifier_nb = GaussianNB()
classifier_nb.fit(X_train, y_train)

end_clock = time.clock()

clock_fit_nb = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_nb, 2), ' sec', sep='')

#----------------------------------------
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_nb = classifier_nb.predict(X_test)

end_clock = time.clock()

clock_predict_nb = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_nb, 2), ' sec', sep='')

#----------------------------------------
# EVALUATE MODEL

# Confusion matrix
cm_nb = confusion_matrix(y_test, y_pred_nb)

# Classification report
cr_nb = classification_report(y_test, y_pred_nb)

print(cm_nb)
print("\n")
print(cr_nb)

acc_nb = cm_nb.diagonal().sum() / cm_nb.sum()
acc_nb
```

```
#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

start_clock = time.clock()

accuracies_nb = cross_val_score(
    estimator=classifier_nb, X=X_train, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_nb = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_nb, 2), ' sec', sep='')

print("Accuracies:")
print(accuracies_nb)
print('\n')
print("RESULTS:")
print(f"  - Mean accuracy: {round(accuracies_nb.mean(), 2)*100}%")
print(f"  - Accuracy std dev: {round(accuracies_nb.std(), 2)*100}%")

df_results_nb = pd.DataFrame([{
    'model':'Naive Bayes',
    'time_fit':clock_fit_nb, 'time_predict':clock_predict_nb,
    'time_10_fold_CV':clock_10FCV_nb,
    'accuracy':acc_nb, 'acc_10_fold':accuracies_nb.mean()}])

df_results_nb = df_results_nb[['model','time_fit','time_predict','time_10_fold_C
V','accuracy','acc_10_fold']]
df_results = df_results_nb
```

A1.4.3 Logistic regression

```
#==============================================================================
#
# LOGISTIC REGRESSION
#
#==============================================================================

#------------------------------------------
# FIT MODEL

start_clock = time.clock()

# Logistic regression
classifier_LogReg = LogisticRegression(random_state=101)
classifier_LogReg.fit(X_train, y_train)

end_clock = time.clock()

clock_fit_LogReg = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_LogReg, 2), ' sec', sep='')

#------------------------------------------
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_LogReg = classifier_LogReg.predict(X_test)

end_clock = time.clock()

clock_predict_LogReg = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_LogReg, 2), ' sec', sep='')

#------------------------------------------
# EVALUATE MODEL

# Confusion matrix
cm_LogReg = confusion_matrix(y_test, y_pred_LogReg)

# Classification report
cr_LogReg = classification_report(y_test, y_pred_LogReg)

print(cm_LogReg)
print("\n")
print(cr_LogReg)

acc_LogReg = cm_LogReg.diagonal().sum() / cm_LogReg.sum()
```

```
#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

start_clock = time.clock()

accuracies_LogReg = cross_val_score(
    estimator=classifier_LogReg, X=X_train, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_LogReg = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_LogReg, 2), ' sec', sep='')

print("Accuracies:")
print(accuracies_LogReg)
print('\n')
print("RESULTS:")
print(f"  - Mean accuracy: {round(accuracies_LogReg.mean(), 2)*100}%")
print(f"  - Accuracy std dev: {round(accuracies_LogReg.std(), 2)*100}%")

df_results_LogReg = pd.DataFrame([{
    'model':'Logistic Regression',
    'time_fit':clock_fit_LogReg, 'time_predict':clock_predict_LogReg,
    'time_10_fold_CV':clock_10FCV_LogReg,
    'accuracy':acc_LogReg, 'acc_10_fold':accuracies_LogReg.mean()}])
df_results_LogReg = df_results_LogReg[['model','time_fit','time_predict','time_10_f
old_CV','accuracy','acc_10_fold']]

df_results = pd.DataFrame.append(df_results, df_results_LogReg).reset_index(drop=Tr
ue)
```

A1.4.4 K nearest neighbors

```
#==============================================================================
#
# K NEAREST NEIGHBORS
#
#==============================================================================

#----------------------------------------
# FIT MODEL

start_clock = time.clock()

# KNN
classifier_knn = KNeighborsClassifier(n_neighbors=5, metric="minkowski", p=2)
classifier_knn.fit(X_train, y_train)

end_clock = time.clock()

clock_fit_knn = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_knn, 2), ' sec', sep='')

#----------------------------------------
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_knn = classifier_knn.predict(X_test)

end_clock = time.clock()

clock_predict_knn = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_knn, 2), ' sec', sep='')

#----------------------------------------
# EVALUATE MODEL

# Confusion matrix
cm_knn = confusion_matrix(y_test, y_pred_knn)

# Classification report
cr_knn = classification_report(y_test, y_pred_knn)

print(cm_knn)
print("\n")
print(cr_knn)

acc_knn = cm_knn.diagonal().sum() / cm_knn.sum()
acc_knn
```

```
#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

# 10-fold cross validation time estimate:
print('10-fold CV estimated time: ',
      round((clock_fit_knn + clock_predict_knn)*10/60, 2),
      ' min')

start_clock = time.clock()

accuracies_knn = cross_val_score(
    estimator=classifier_knn, X=X_train, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_knn = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_knn, 2), ' sec', sep='')

print("Accuracies:")
print(accuracies_knn)
print('\n')
print("RESULTS:")
print(f"  - Mean accuracy: {round(accuracies_knn.mean(), 2)*100}%")
print(f"  - Accuracy std dev: {round(accuracies_knn.std(), 2)*100}%")

df_results_knn = pd.DataFrame([{
    'model':'K Nearest Neighbors',
    'time_fit':clock_fit_knn, 'time_predict':clock_predict_knn,
    'time_10_fold_CV':clock_10FCV_knn,
    'accuracy':acc_knn, 'acc_10_fold':accuracies_knn.mean()}])
df_results_knn = df_results_knn[['model','time_fit','time_predict','time_10_fold_C
V','accuracy','acc_10_fold']]

df_results = pd.DataFrame.append(df_results, df_results_knn).reset_index(drop=True)
```

A1.4.5 Support vector machine

```
#==============================================================================
#
# SUPPORT VECTOR MACHINE
#
#==============================================================================

#----------------------------------------
# FIT MODEL

start_clock = time.clock()

# SVM
classifier_svm_linear = SVC(kernel="linear", random_state=101)
classifier_svm_linear.fit(X_train, y_train)

end_clock = time.clock()

clock_fit_svm_linear = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_svm_linear, 2), ' sec', sep='')

#----------------------------------------
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_svm_linear = classifier_svm_linear.predict(X_test)

end_clock = time.clock()

clock_predict_svm_linear = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_svm_linear, 2), ' sec', sep='')

#----------------------------------------
# EVALUATE MODEL

# Confusion matrix
cm_svm_linear = confusion_matrix(y_test, y_pred_svm_linear)

# Classification report
cr_svm_linear = classification_report(y_test, y_pred_svm_linear)

print(cm_svm_linear)
print("\n")
print(cr_svm_linear)

acc_svm_linear = cm_svm_linear.diagonal().sum() / cm_svm_linear.sum()
acc_svm_linear
```

```
#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

# 10-fold cross validation time estimate:
print('10-fold CV estimated time: ',
      round((clock_fit_svm_linear + clock_predict_svm_linear)*10/60/60, 2),
      ' hours')

# *** TIME-INTENSIVE ***
"""start_clock = time.clock()

accuracies_svm = cross_val_score(
    estimator=classifier_svm, X=X_train, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_svm = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_svm, 2), ' sec', sep='')"""

try:
    clock_10FCV_svm_linear
except:
    clock_10FCV_svm_linear = None
    print("No 10-fold CV time to report")

try:
    accuracies_svm_linear
except:
    accuracy_10FCV_mean_svm_linear = None
    print("No K-fold CV accuracies to report")
else:
    accuracy_10FCV_mean_svm_linear = accuracies_svm_linear.mean()
    print("Accuracies:")
    print(accuracies_svm_linear)
    print('\n')
    print("RESULTS:")
    print(f"  - Mean accuracy: {round(accuracies_svm_linear.mean(),4)*100}%")
    print(f"  - Accuracy std dev: {round(accuracies_svm_linear.std(),4)*100}%")

df_results_svm_linear = pd.DataFrame([{
    'model':'SVM, Linear',
    'time_fit':clock_fit_svm_linear, 'time_predict':clock_predict_svm_linear,
    'time_10_fold_CV':clock_10FCV_svm_linear,
    'accuracy':acc_svm_linear, 'acc_10_fold':accuracy_10FCV_mean_svm_linear}])
df_results_svm_linear = df_results_svm_linear[['model','time_fit','time_predict','t
ime_10_fold_CV','accuracy','acc_10_fold']]
```

```
df_results = pd.DataFrame.append(df_results, df_results_svm_linear).reset_index(dro
p=True)
```

A1.4.6 Support vector machine, kernel RBF

```
#==============================================================================
#
# SUPPORT VECTOR MACHINE, KERNEL RBF
#
#==============================================================================

#----------------------------------------
# FIT MODEL

start_clock = time.clock()

# SVM
classifier_svm_rbf = SVC(kernel="rbf", random_state=101)
classifier_svm_rbf.fit(X_train, y_train)

end_clock = time.clock()

clock_fit_svm_rbf = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_svm_rbf, 2), ' sec', sep='')

#----------------------------------------
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_svm_rbf = classifier_svm_rbf.predict(X_test)

end_clock = time.clock()

clock_predict_svm_rbf = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_svm_rbf, 2), ' sec', sep='')

#----------------------------------------
# EVALUATE MODEL

# Confusion matrix
cm_svm_rbf = confusion_matrix(y_test, y_pred_svm_rbf)

# Classification report
cr_svm_rbf = classification_report(y_test, y_pred_svm_rbf)

print(cm_svm_rbf)
print("\n")
print(cr_svm_rbf)

acc_svm_rbf = cm_svm_rbf.diagonal().sum() / cm_svm_rbf.sum()
acc_svm_rbf
```

```
#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

# 10-fold cross validation time estimate:
print('10-fold CV estimated time: ',
      round((clock_fit_svm_rbf + clock_predict_svm_rbf)*10/60/60, 2),
      ' hours')

# *** TIME-INTENSIVE ***
"""start_clock = time.clock()

accuracies_svm_rbf = cross_val_score(
    estimator=classifier_svm_rbf, X=X_train, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_svm_rbf = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_svm_rbf, 2), ' sec', sep='')"""

try:
    clock_10FCV_svm_rbf
except:
    clock_10FCV_svm_rbf = None
    print("No 10-fold CV time to report")

try:
    accuracies_svm_rbf
except:
    accuracy_10FCV_mean_svm_rbf = None
    print("No K-fold CV accuracies to report")
else:
    accuracy_10FCV_mean_svm_rbf = accuracies_svm_rbf.mean()
    print("Accuracies:")
    print(accuracies_svm_rbf)
    print('\n')
    print("RESULTS:")
    print(f"  - Mean accuracy: {round(accuracies_svm_rbf.mean(), 2)*100}%")
    print(f"  - Accuracy std dev: {round(accuracies_svm_rbf.std(), 2)*100}%")

df_results_svm_rbf = pd.DataFrame([{
    'model':'SVM, RBF',
    'time_fit':clock_fit_svm_rbf, 'time_predict':clock_predict_svm_rbf,
    'time_10_fold_CV':clock_10FCV_svm_rbf,
    'accuracy':acc_svm_rbf, 'acc_10_fold':accuracy_10FCV_mean_svm_rbf}])
df_results_svm_rbf = df_results_svm_rbf[['model','time_fit','time_predict','time_10
_fold_CV','accuracy','acc_10_fold']]
```

```
df_results = pd.DataFrame.append(df_results, df_results_svm_rbf).reset_index(drop=T
rue)
```

A1.4.7 Decision tree

```
#==============================================================================
#
# DECISION TREE CLASSIFICATION
#
#==============================================================================

#---------------------------------------
# FIT MODEL

start_clock = time.clock()

# Decision tree
classifier_dt = DecisionTreeClassifier(criterion="entropy", random_state=101)
classifier_dt.fit(X_train, y_train)

end_clock = time.clock()

clock_fit_dt = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_dt, 2), ' sec', sep='')

#---------------------------------------
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_dt = classifier_dt.predict(X_test)

end_clock = time.clock()

clock_predict_dt = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_dt, 2), ' sec', sep='')

#---------------------------------------
# EVALUATE MODEL

# Confusion matrix
cm_dt = confusion_matrix(y_test, y_pred_dt)

# Classification report
cr_dt = classification_report(y_test, y_pred_dt)

print(cm_dt)
print("\n")
print(cr_dt)

acc_dt = cm_dt.diagonal().sum() / cm_dt.sum()
acc_dt
```

```
#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

# 10-fold cross validation time estimate:
print('10-fold CV estimated time: ',
      round((clock_fit_dt + clock_predict_dt)*10, 2),
      ' sec')

start_clock = time.clock()

accuracies_dt = cross_val_score(
    estimator=classifier_dt, X=X_train, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_dt = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_dt, 2), ' sec', sep='')

try:
    clock_10FCV_dt
except:
    clock_10FCV_dt = None
    print("No 10-fold CV time to report")

try:
    accuracies_dt
except:
    accuracy_10FCV_mean_dt = None
    print("No K-fold CV accuracies to report")
else:
    accuracy_10FCV_mean_dt = accuracies_dt.mean()
    print("Accuracies:")
    print(accuracies_dt)
    print('\n')
    print("RESULTS:")
    print(f"  - Mean accuracy: {round(accuracies_dt.mean(), 2)*100}%")
    print(f"  - Accuracy std dev: {round(accuracies_dt.std(), 2)*100}%")

df_results_dt = pd.DataFrame([{
    'model':'Decision Tree',
    'time_fit':clock_fit_dt, 'time_predict':clock_predict_dt,
    'time_10_fold_CV':clock_10FCV_dt,
    'accuracy':acc_dt, 'acc_10_fold':accuracy_10FCV_mean_dt}])
df_results_dt = df_results_dt[['model','time_fit','time_predict','time_10_fold_C
V','accuracy','acc_10_fold']]

df_results = pd.DataFrame.append(df_results, df_results_dt).reset_index(drop=True)
```

A1.4.8 Random forest, 10-fold

```
#==============================================================================
#
# RANDOM FOREST CLASSIFICATION (10 fold)
#
#==============================================================================

#-----------------------------------------
# FIT MODEL

start_clock = time.clock()

# Decision tree
classifier_rf10 = RandomForestClassifier(n_estimators=10, criterion="entropy", rand
om_state=101)
classifier_rf10.fit(X_train, y_train)

end_clock = time.clock()

clock_fit_rf10 = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_rf10, 2), ' sec', sep='')

#-----------------------------------------
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_rf10 = classifier_rf10.predict(X_test)

end_clock = time.clock()

clock_predict_rf10 = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_rf10, 2), ' sec', sep='')

#-----------------------------------------
# EVALUATE MODEL

# Confusion matrix
cm_rf10 = confusion_matrix(y_test, y_pred_rf10)

# Classification report
cr_rf10 = classification_report(y_test, y_pred_rf10)

print(cm_rf10)
print("\n")
print(cr_rf10)

acc_rf10 = cm_rf10.diagonal().sum() / cm_rf10.sum()
```

```
acc_rf10

#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

# 10-fold cross validation time estimate:
print('10-fold CV estimated time: ',
      round((clock_fit_rf10 + clock_predict_rf10)*10, 2),
      ' sec')

start_clock = time.clock()

accuracies_rf10 = cross_val_score(
    estimator=classifier_rf10, X=X_train, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_rf10 = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_rf10, 2), ' sec', sep='')

try:
    clock_10FCV_rf10
except:
    clock_10FCV_rf10 = None
    print("No 10-fold CV time to report")

try:
    accuracies_rf10
except:
    accuracy_10FCV_mean_rf10 = None
    print("No K-fold CV accuracies to report")
else:
    accuracy_10FCV_mean_rf10 = accuracies_rf10.mean()
    print("Accuracies:")
    print(accuracies_rf10)
    print('\n')
    print("RESULTS:")
    print(f"  - Mean accuracy: {round(accuracies_rf10.mean(), 2)*100}%")
    print(f"  - Accuracy std dev: {round(accuracies_rf10.std(), 2)*100}%")

df_results_rf10 = pd.DataFrame([{
    'model':'Random Forest (10-fold)',
    'time_fit':clock_fit_rf10, 'time_predict':clock_predict_rf10,
    'time_10_fold_CV':clock_10FCV_rf10,
    'accuracy':acc_rf10, 'acc_10_fold':accuracy_10FCV_mean_rf10}])
df_results_rf10 = df_results_rf10[['model','time_fit','time_predict','time_10_fold_
CV','accuracy','acc_10_fold']]
```

```
df_results = pd.DataFrame.append(df_results, df_results_rf10).reset_index(drop=Tru
e)
```

A1.4.9 Principal component analysis

```
#==============================================================================
#
# PRINCIPAL COMPONENT ANALYSIS
#
#==============================================================================

# Explore principal components
pca = PCA(n_components=None, random_state=101)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
explained_variance_pca = pca.explained_variance_ratio_

print(explained_variance_pca)

plt.figure(figsize=(8,6))
plt.scatter(X_train_pca[:,0], X_train_pca[:,1], c=y_train, cmap='plasma')
plt.xlabel('First principal component')
plt.ylabel('Second Principal Component')

# Explore top two principal components
pca = PCA(n_components=2, random_state=101)
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.transform(X_test)
explained_variance_pca = pca.explained_variance_ratio_

plt.figure(figsize=(8,6))
plt.scatter(X_train_pca[:,0], X_train_pca[:,1], c=y_train, cmap='plasma')
plt.xlabel('First principal component')
plt.ylabel('Second Principal Component')

#--------------------------------------
# FIT MODEL (NAIVE BAYES)

start_clock = time.clock()

# Naive-Bayes
classifier_pca2_nb = GaussianNB()
classifier_pca2_nb.fit(X_train_pca, y_train)

end_clock = time.clock()

clock_fit_pca2_nb = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_pca2_nb, 2), ' sec', sep='')

end_clock = time.clock()

#--------------------------------------
```

```
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_pca2_nb = classifier_pca2_nb.predict(X_test_pca)

end_clock = time.clock()

clock_predict_pca2_nb = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_pca2_nb, 2), ' sec', sep='')

#----------------------------------------
# EVALUATE MODEL

# Confusion matrix
cm_pca2_nb = confusion_matrix(y_test, y_pred_pca2_nb)

# Classification report
cr_pca2_nb = classification_report(y_test, y_pred_pca2_nb)

print(cm_pca2_nb)
print("\n")
print(cr_pca2_nb)

acc_pca2_nb = cm_pca2_nb.diagonal().sum() / cm_pca2_nb.sum()
acc_pca2_nb

#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

start_clock = time.clock()

accuracies_pca2_nb = cross_val_score(
    estimator=classifier_pca2_nb, X=X_train_pca, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_pca2_nb = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_pca2_nb, 2), ' sec', sep='')

try:
    clock_10FCV_pca2_nb
except:
    clock_10FCV_pca2_nb = None
    print("No 10-fold CV time to report")

try:
    accuracies_pca2_nb
```

```
except:
    accuracy_10FCV_mean_pca2_nb = None
    print("No K-fold CV accuracies to report")
else:
    accuracy_10FCV_mean_pca2_nb = accuracies_pca2_nb.mean()
    print("Accuracies:")
    print(accuracies_pca2_nb)
    print('\n')
    print("RESULTS:")
    print(f"  - Mean accuracy: {round(accuracies_pca2_nb.mean(), 2)*100}%")
    print(f"  - Accuracy std dev: {round(accuracies_pca2_nb.std(), 2)*100}%")

df_results_pca2_nb = pd.DataFrame([{
    'model':'PCA (n=2), Naive Bayes',
    'time_fit':clock_fit_pca2_nb, 'time_predict':clock_predict_pca2_nb,
    'time_10_fold_CV':clock_10FCV_pca2_nb,
    'accuracy':acc_pca2_nb, 'acc_10_fold':accuracy_10FCV_mean_pca2_nb}])
df_results_pca2_nb = df_results_pca2_nb[['model','time_fit','time_predict','time_10
_fold_CV','accuracy','acc_10_fold']]

df_results = pd.DataFrame.append(df_results, df_results_pca2_nb).reset_index(drop=T
rue)
```

A1.5 Model tuning and selection (as of 2018-12-12)

Relevant file: '04 Working - Models.ipynb'

A1.5.1 Grid search - random forest

```
#================================================================================
#
# GRID SEARCH - RANDOM FOREST
#
#================================================================================

# FIT MODEL
classifier_grid_rf = RandomForestClassifier(random_state=101)

# Decide parameters to loop through
parameters_rf = {
    'max_features': [None, 'sqrt', 'log2'],
    'n_estimators': [1,5,10,20,30,50,75,100],
    'min_samples_leaf': [1,2,5,10,25,50,100],
    'criterion': ['gini', 'entropy']
}

# Create validation curve objects
param_range = np.arange(1,110,10)
train_scores, test_scores = validation_curve(
    RandomForestClassifier(random_state=101), X_train, y_train, param_name='n_estim
ators',
    param_range=param_range, cv=5, scoring="accuracy", n_jobs=-1)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Plot validation curve
plt.title("Validation Curve with Random Forest")
plt.xlabel("n_neighbors")
plt.ylabel("Score")
#plt.ylim(0.68, 0.72)
lw = 2
plt.semilogx(param_range, train_scores_mean, label="Training score",
             color="darkorange", lw=lw)
plt.fill_between(param_range, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.2,
                 color="darkorange", lw=lw)
plt.semilogx(param_range, test_scores_mean, label="Cross-validation score",
             color="navy", lw=lw)
plt.fill_between(param_range, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.2,
                 color="navy", lw=lw)
plt.legend(loc="best")
plt.show()
```

```
# Create grid search object
grid_search_rf = GridSearchCV(estimator=classifier_grid_rf,
    param_grid=parameters_rf, scoring="accuracy", cv=5,
    n_jobs=-1, verbose=10)

# Fit grid search object to training set

start_clock = time.clock()
grid_search_rf = grid_search_rf.fit(X_train, y_train)
end_clock = time.clock()

clock_grid_search_rf = end_clock - start_clock

print('Runtime, grid search: ', round(clock_grid_search_rf, 2), ' sec', sep='')

# Print results
best_accuracy_grid_search_rf = grid_search_rf.best_score_
best_parameters_grid_search_rf = grid_search_rf.best_params_
print(f"Best accuracy: {round(best_accuracy_grid_search_rf, 2)*100}%")
print("\n")
print(f"Best parameters: {best_parameters_grid_search_rf}")

# Search again

# Decide parameters to loop through
parameters_rf = {
    'max_features': ['sqrt'],
    'n_estimators': [100,200,300],
    'min_samples_leaf': [15,20,25,30,35,40,45],
    'criterion': ['gini']
}

# Create grid search object
grid_search_rf = GridSearchCV(estimator=classifier_grid_rf,
    param_grid=parameters_rf, scoring="accuracy", cv=5,
    n_jobs=-1, verbose=10)

# Fit grid search object to training set
start_clock = time.clock()
grid_search_rf = grid_search_rf.fit(X_train, y_train)
end_clock = time.clock()
clock_grid_search_rf = end_clock - start_clock
print('Runtime, grid search: ', round(clock_grid_search_rf, 2), ' sec', sep='')

# Print results
best_accuracy_grid_search_rf = grid_search_rf.best_score_
best_parameters_grid_search_rf = grid_search_rf.best_params_
print(f"Best accuracy: {round(best_accuracy_grid_search_rf, 2)*100}%")
print("\n")
```

```
print(f"Best parameters: {best_parameters_grid_search_rf}")

# Search again

# Decide parameters to loop through
parameters_rf = {
    'max_features': ['sqrt'],
    'n_estimators': [90,100,110],
    'min_samples_leaf': [21,23,25,27,29],
    'criterion': ['gini']
}

# Create grid search object
grid_search_rf = GridSearchCV(estimator=classifier_grid_rf,
    param_grid=parameters_rf, scoring="accuracy", cv=5,
    n_jobs=-1, verbose=10)

# Fit grid search object to training set
start_clock = time.clock()
grid_search_rf = grid_search_rf.fit(X_train, y_train)
end_clock = time.clock()
clock_grid_search_rf = end_clock - start_clock
print('Runtime, grid search: ', round(clock_grid_search_rf, 2), ' sec', sep='')

# Print results
best_accuracy_grid_search_rf = grid_search_rf.best_score_
best_parameters_grid_search_rf = grid_search_rf.best_params_
print(f"Best accuracy: {round(best_accuracy_grid_search_rf, 2)*100}%")
print("\n")
print(f"Best parameters: {best_parameters_grid_search_rf}")

# ---- RUN WITH OPTIMIZED PARAMETERS ----

#----------------------------------------
# FIT MODEL

start_clock = time.clock()

# Decision tree
classifier_rf_opt = RandomForestClassifier(
    n_estimators=100, criterion="gini", max_features='sqrt', min_samples_leaf=25, r
andom_state=101)
classifier_rf_opt.fit(X_train, y_train)

end_clock = time.clock()

clock_fit_rf_opt = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_rf_opt, 2), ' sec', sep='')
```

```
#----------------------------------------
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_rf_opt = classifier_rf_opt.predict(X_test)

end_clock = time.clock()

clock_predict_rf_opt = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_rf_opt, 2), ' sec', sep='')

#----------------------------------------
# EVALUATE MODEL

# Confusion matrix
cm_rf_opt = confusion_matrix(y_test, y_pred_rf_opt)

# Classification report
cr_rf_opt = classification_report(y_test, y_pred_rf_opt)

print(cm_rf_opt)
print("\n")
print(cr_rf_opt)

acc_rf_opt = cm_rf_opt.diagonal().sum() / cm_rf_opt.sum()
acc_rf_opt

#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

# 10-fold cross validation time estimate:
print('10-fold CV estimated time: ',
      round((clock_fit_rf_opt + clock_predict_rf_opt)*10, 2),
      ' sec')

start_clock = time.clock()

accuracies_rf_opt = cross_val_score(
    estimator=classifier_rf_opt, X=X_train, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_rf_opt = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_rf_opt, 2), ' sec', sep='')

try:
    clock_10FCV_rf_opt
```

```
except:
    clock_10FCV_rf_opt = None
    print("No 10-fold CV time to report")

try:
    accuracies_rf_opt
except:
    accuracy_10FCV_mean_rf_opt = None
    print("No K-fold CV accuracies to report")
else:
    accuracy_10FCV_mean_rf_opt = accuracies_rf_opt.mean()
    print("Accuracies:")
    print(accuracies_rf_opt)
    print('\n')
    print("RESULTS:")
    print(f"  - Mean accuracy: {round(accuracies_rf_opt.mean(), 2)*100}%")
    print(f"  - Accuracy std dev: {round(accuracies_rf_opt.std(), 2)*100}%")

df_results_rf_opt = pd.DataFrame([{
    'model':'Random Forest (Optimized)',
    'time_fit':clock_fit_rf_opt, 'time_predict':clock_predict_rf_opt,
    'time_10_fold_CV':clock_10FCV_rf_opt,
    'accuracy':acc_rf_opt, 'acc_10_fold':accuracy_10FCV_mean_rf_opt}])
df_results_rf_opt = df_results_rf_opt[['model','time_fit','time_predict','time_10_f
old_CV','accuracy','acc_10_fold']]

df_results = pd.DataFrame.append(df_results, df_results_rf_opt).reset_index(drop=Tr
ue)
```

A1.5.2 Grid search - logistic regression

```
#==============================================================================
#
# GRID SEARCH - LOGISTIC REGRESSION
#
#==============================================================================

# FIT MODEL
classifier_grid_LogReg = LogisticRegression(random_state=101)

# Decide parameters to loop through
param_range = np.logspace(0,3,10)
parameters_LogReg = {
    'penalty': ['l1','l2'],
    'C': param_range
}

# Create grid search object
grid_search_LogReg = GridSearchCV(estimator=classifier_grid_LogReg,
    param_grid=parameters_LogReg, scoring="accuracy", cv=5,
    n_jobs=-1, verbose=10)

# Create validation curve objects
train_scores, test_scores = validation_curve(
    LogisticRegression(random_state=101), X_train, y_train, param_name='C',
    param_range=param_range, cv=5, scoring="accuracy", n_jobs=-1)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Plot validation curve
plt.title("Validation Curve with Logistic Regression")
plt.xlabel("C")
plt.ylabel("Score")
plt.ylim(0.68, 0.70)
lw = 2
plt.semilogx(param_range, train_scores_mean, label="Training score",
             color="darkorange", lw=lw)
plt.fill_between(param_range, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.2,
                 color="darkorange", lw=lw)
plt.semilogx(param_range, test_scores_mean, label="Cross-validation score",
             color="navy", lw=lw)
plt.fill_between(param_range, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.2,
                 color="navy", lw=lw)
plt.legend(loc="best")
```

```
plt.show()

# Fit grid search object to training set

start_clock = time.clock()
grid_search_LogReg = grid_search_LogReg.fit(X_train, y_train)
end_clock = time.clock()

clock_grid_search_LogReg = end_clock - start_clock

print('Runtime, grid search: ', round(clock_grid_search_LogReg, 2), ' sec', sep='')

# Print results
best_accuracy_grid_search_LogReg = grid_search_LogReg.best_score_
best_parameters_grid_search_LogReg = grid_search_LogReg.best_params_
print(f"Best accuracy: {round(best_accuracy_grid_search_LogReg, 2)*100}%")
print("\n")
print(f"Best parameters: {best_parameters_grid_search_LogReg}")

# Search again

# Decide parameters to loop through
parameters_LogReg = {
    'penalty': ['l1'],
    'C': np.logspace(0.5,2,19)
}

# Create grid search object
grid_search_LogReg = GridSearchCV(estimator=classifier_grid_LogReg,
    param_grid=parameters_LogReg, scoring="accuracy", cv=5,
    n_jobs=-1, verbose=10)

# Fit grid search object to training set
start_clock = time.clock()
grid_search_LogReg = grid_search_LogReg.fit(X_train, y_train)
end_clock = time.clock()
clock_grid_search_LogReg = end_clock - start_clock
print('Runtime, grid search: ', round(clock_grid_search_LogReg, 2), ' sec', sep='')

# Print results
best_accuracy_grid_search_LogReg = grid_search_LogReg.best_score_
best_parameters_grid_search_LogReg = grid_search_LogReg.best_params_
print(f"Best accuracy: {round(best_accuracy_grid_search_LogReg, 2)*100}%")
print("\n")
print(f"Best parameters: {best_parameters_grid_search_LogReg}")

# ---- RUN WITH OPTIMIZED PARAMETERS ----

#----------------------------------------
```

```
# FIT MODEL

start_clock = time.clock()

classifier_LogReg_opt = LogisticRegression(random_state=101, C=10.0, penalty='l1')
classifier_LogReg_opt.fit(X_train, y_train)

end_clock = time.clock()

clock_fit_LogReg_opt = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_LogReg_opt, 2), ' sec', sep='')

#----------------------------------------
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_LogReg_opt = classifier_LogReg_opt.predict(X_test)

end_clock = time.clock()

clock_predict_LogReg_opt = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_LogReg_opt, 2), ' sec', sep='')

#----------------------------------------
# EVALUATE MODEL

# Confusion matrix
cm_LogReg_opt = confusion_matrix(y_test, y_pred_LogReg_opt)

# Classification report
cr_LogReg_opt = classification_report(y_test, y_pred_LogReg_opt)

print(cm_LogReg_opt)
print("\n")
print(cr_LogReg_opt)

acc_LogReg_opt = cm_LogReg_opt.diagonal().sum() / cm_LogReg_opt.sum()
acc_LogReg_opt

#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

# 10-fold cross validation time estimate:
print('10-fold CV estimated time: ',
      round((clock_fit_LogReg_opt + clock_predict_LogReg_opt)*10, 2),
      ' sec')

start_clock = time.clock()
```

```
accuracies_LogReg_opt = cross_val_score(
    estimator=classifier_LogReg_opt, X=X_train, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_LogReg_opt = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_LogReg_opt, 2), ' sec', sep='')

try:
    clock_10FCV_LogReg_opt
except:
    clock_10FCV_LogReg_opt = None
    print("No 10-fold CV time to report")

try:
    accuracies_LogReg_opt
except:
    accuracy_10FCV_mean_LogReg_opt = None
    print("No K-fold CV accuracies to report")
else:
    accuracy_10FCV_mean_LogReg_opt = accuracies_LogReg_opt.mean()
    print("Accuracies:")
    print(accuracies_LogReg_opt)
    print('\n')
    print("RESULTS:")
    print(f"  - Mean accuracy: {round(accuracies_LogReg_opt.mean(), 2)*100}%")
    print(f"  - Accuracy std dev: {round(accuracies_LogReg_opt.std(), 2)*100}%")

df_results_LogReg_opt = pd.DataFrame([{
    'model':'Logistic Regression (Optimized)',
    'time_fit':clock_fit_LogReg_opt, 'time_predict':clock_predict_LogReg_opt,
    'time_10_fold_CV':clock_10FCV_LogReg_opt,
    'accuracy':acc_LogReg_opt, 'acc_10_fold':accuracy_10FCV_mean_LogReg_opt}])
df_results_LogReg_opt = df_results_LogReg_opt[['model','time_fit','time_predict','t
ime_10_fold_CV','accuracy','acc_10_fold']]

df_results = pd.DataFrame.append(df_results, df_results_LogReg_opt).reset_index(dro
p=True)
```

A1.5.3 Grid search - K nearest neighbors

```
#==============================================================================
#
# GRID SEARCH - K NEAREST NEIGHBORS
#
#==============================================================================

classifier_grid_knn = KNeighborsClassifier()

# Decide parameters to loop through
param_range = np.arange(1,25,1)
parameters_knn = {
    'n_neighbors': param_range,
    'metric': ['minkowski'],
    'p': [1, 2]
}

# Create validation curve objects
train_scores, test_scores = validation_curve(
    KNeighborsClassifier(), X_train, y_train, param_name='n_neighbors',
    param_range=param_range, cv=5, scoring="accuracy", n_jobs=-1)
train_scores_mean = np.mean(train_scores, axis=1)
train_scores_std = np.std(train_scores, axis=1)
test_scores_mean = np.mean(test_scores, axis=1)
test_scores_std = np.std(test_scores, axis=1)

# Plot validation curve
plt.title("Validation Curve with K Nearest Neighbors")
plt.xlabel("n_neighbors")
plt.ylabel("Score")
#plt.ylim(0.68, 0.72)
lw = 2
plt.semilogx(param_range, train_scores_mean, label="Training score",
             color="darkorange", lw=lw)
plt.fill_between(param_range, train_scores_mean - train_scores_std,
                 train_scores_mean + train_scores_std, alpha=0.2,
                 color="darkorange", lw=lw)
plt.semilogx(param_range, test_scores_mean, label="Cross-validation score",
             color="navy", lw=lw)
plt.fill_between(param_range, test_scores_mean - test_scores_std,
                 test_scores_mean + test_scores_std, alpha=0.2,
                 color="navy", lw=lw)
plt.legend(loc="best")
plt.show()

# Create grid search object
grid_search_knn = GridSearchCV(estimator=classifier_grid_knn,
    param_grid=parameters_knn, scoring="accuracy", cv=5,
```

```
    n_jobs=-1, verbose=10)

# Fit grid search object to training set

start_clock = time.clock()
grid_search_knn = grid_search_knn.fit(X_train, y_train)
end_clock = time.clock()

clock_grid_search_knn = end_clock - start_clock

print('Runtime, grid search: ', round(clock_grid_search_knn, 2), ' sec', sep='')

# Print results
best_accuracy_grid_search_knn = grid_search_knn.best_score_
best_parameters_grid_search_knn = grid_search_knn.best_params_
print("Best accuracy: ", round(best_accuracy_grid_search_knn, 2)*100, "%", sep="")
print("\n")
print("Best parameters: ", best_parameters_grid_search_knn, sep="")

# ---- RUN WITH OPTIMIZED PARAMETERS ----

#----------------------------------------
# FIT MODEL

start_clock = time.clock()

classifier_knn_opt = KNeighborsClassifier(metric='minkowski', n_neighbors=23, p=2,
 n_jobs=-1)
classifier_knn_opt.fit(X_train, y_train)

end_clock = time.clock()

clock_fit_knn_opt = end_clock - start_clock
print('Runtime, fit: ', round(clock_fit_knn_opt, 2), ' sec', sep='')

#----------------------------------------
# PREDICT TEST RESULTS

start_clock = time.clock()

y_pred_knn_opt = classifier_knn_opt.predict(X_test)

end_clock = time.clock()

clock_predict_knn_opt = end_clock - start_clock
print('Runtime, predict: ', round(clock_predict_knn_opt, 2), ' sec', sep='')

#----------------------------------------
# EVALUATE MODEL
```

```
# Confusion matrix
cm_knn_opt = confusion_matrix(y_test, y_pred_knn_opt)

# Classification report
cr_knn_opt = classification_report(y_test, y_pred_knn_opt)

print(cm_knn_opt)
print("\n")
print(cr_knn_opt)

acc_knn_opt = cm_knn_opt.diagonal().sum() / cm_knn_opt.sum()
acc_knn_opt

#----------------------------------------
# APPLY K-FOLD CROSS VALIDATION

# 10-fold cross validation time estimate:
print('10-fold CV estimatknnd time: ',
      round((clock_fit_knn_opt + clock_predict_knn_opt)*10, 2),
      ' sec')

start_clock = time.clock()

accuracies_knn_opt = cross_val_score(
    estimator=classifier_knn_opt, X=X_train, y=y_train,
    cv=10)

end_clock = time.clock()

clock_10FCV_knn_opt = end_clock - start_clock
print('Runtime, 10-fold CV: ', round(clock_10FCV_knn_opt, 2), ' sec', sep='')

try:
    clock_10FCV_knn_opt
except:
    clock_10FCV_knn_opt = None
    print("No 10-fold CV time to report")
    try:
    accuracies_knn_opt
except:
    accuracy_10FCV_mean_knn_opt = None
    print("No K-fold CV accuracies to report")
else:
    accuracy_10FCV_mean_knn_opt = accuracies_knn_opt.mean()
    print("Accuracies:")
    print(accuracies_knn_opt)
    print('\n')
    print("RESULTS:")
```

```
    print("  - Mean accuracy: ", round(accuracies_knn_opt.mean(), 2)*100, "%", sep
="")
    print("  - Accuracy std dev: ", round(accuracies_knn_opt.std(), 2)*100, "%", se
p="")

df_results_knn_opt = pd.DataFrame([{
    'model':'KNN (Optimized)',
    'time_fit':clock_fit_knn_opt, 'time_predict':clock_predict_knn_opt,
    'time_10_fold_CV':clock_10FCV_knn_opt,
    'accuracy':acc_knn_opt, 'acc_10_fold':accuracy_10FCV_mean_knn_opt}])
df_results_knn_opt = df_results_knn_opt[['model','time_fit','time_predict','time_10
_fold_CV','accuracy','acc_10_fold']]

df_results = pd.DataFrame.append(df_results, df_results_knn_opt).reset_index(drop=T
rue)
```

A2 Code - new data prediction

A2.1 Predict function

Relevant file: 'f_predict.py'

```
# -*- coding: utf-8 -*-
"""
Created on Mon Dec 10 14:25:31 2018

@author: steve
"""

#==============================================================================
#
# IMPORT LIBRARIES
#
#==============================================================================
import dill
import time

#==============================================================================
#
# PICKLING
#
#==============================================================================
sc_X = dill.load(open("sc_X.pkl", "rb"))
classifier_rf_opt = dill.load(open("classifier_rf_opt.pkl", "rb"))

#==============================================================================
#
# FUNCTION - PREDICT
#
#==============================================================================
def predictLaunchState(X):
    '''
    This function accepts a new data dataframe and uses a chosen machine
    learning model to predict whether each observation will be successfully
    funded ('launch_state' = 1) or not ('launch_state' = 0).
    '''

    #-----------------------------------------
    # X/y SPLIT

    info_variables = ['id','launched_at','category','country',
                      'pledged_ratio', 'backers_count']
    X = X.drop(columns=info_variables)

    #-----------------------------------------
    # SCALE FEATURES
    X = sc_X.transform(X)

    #-----------------------------------------
```

```
    # PREDICT RESULTS

    start_clock = time.clock()
    y_pred = classifier_rf_opt.predict(X)
    end_clock = time.clock()

    clock_predict = end_clock - start_clock

    print('\n')
    print('Runtime, predict: ', round(clock_predict, 2), ' sec', sep='')

    return y_pred
```

A2.2 Final prediction script

Relevant file: 'predict.py'

```
# -*- coding: utf-8 -*-
"""
Created on Mon Dec 10 16:49:34 2018

@author: steve
"""

#==============================================================================
#
# IMPORT LIBRARIES
#
#==============================================================================
import numpy as np
import pandas as pd
import os
from f_dataImport import dataImport
from f_cleanData import cleanData
from f_predict import predictLaunchState
from sklearn.metrics import confusion_matrix, classification_report

#==============================================================================
#
# RAW DATA IMPORT
#
#==============================================================================

while True:
    print('\n')
    try:
        choice = int(input('What sort of raw data do you have? (Enter 1 or 2): \n'
                           '  [1] JSON data  \n'
                           '  [2] Dataframe from previously cleaned raw JSON file:
 '))
    except:
        print('\n')
        print('Input an integer (1 or 2)')
        continue
    else:
        if (choice == 1):
            df = dataImport()
            break
        elif (choice == 2):
            print('\n')
            new_data = str(input('Input the new dataframe filepath you want to pred
ict: '))

            if not os.path.exists(new_data):
```

```
                print('\n')
                print('The filepath \'', new_data, '\' does not exist.', sep='')
                continue
            else:
                print('\n')
                yesno = str(input(f'Confirm that \'{new_data}\' is the correct '
                                  'filepath (\'y\' or \'n\'): '))

                if (yesno[0].lower() == "y"):
                    df = pd.read_csv(new_data, sep=',', na_filter=False, index_col=
0)
                    break
                elif (yesno[0].lower() == 'n'):
                    print('\n')
                    continue
                else:
                    print('\n')
                    print('Improper input')
                    continue
            break
        else:
            print('\n')
            print('Must input 1 or 2')
            continue

#============================================================================
#
# CLEAN DATA
#
#============================================================================

df = cleanData(df)

#============================================================================
#
# EXTRACT OUTCOME
#
#============================================================================

X = df.drop(columns = 'launch_state')
y = df['launch_state']


#============================================================================
#
# PREDICT
#
#============================================================================
```

```
y_pred = predictLaunchState(X)

#----------------------------------------
# WRITE OUT PREDICTIONS
np.savetxt('y_pred.csv', y_pred, delimiter=',')

#==============================================================================
#
# EVALUATE IF APPLICABLE
#
#==============================================================================

if (y.unique().any() == None):
    print('\n')
    print('This appears to be new data. The prediction vector has been created '
          'and is called \'y_pred\'. A comma-separated value copy has been '
          'saved as \'y_pred.csv\'.')
else:
    #----------------------------------------
    # EVALUATE MODEL

    # Confusion matrix
    cm = confusion_matrix(y, y_pred)

    # Classification report
    cr = classification_report(y, y_pred)

    # Accuracy
    acc = cm.diagonal().sum() / cm.sum()

    print("\n")
    print("CONFUSION MATRIX:", sep="")
    print(cm)
    print("\n")
    print("CLASSIFICATION REPORT:", sep="")
    print(cr)
    print("\n")
    print("ACCURACY: ", round(acc, 2), sep="")
    print('\n')
    print('The prediction vector has been created'
          'and is called \'y_pred\'. A comma-separated value copy has been '
          'saved as \'y_pred.csv\'.')
```