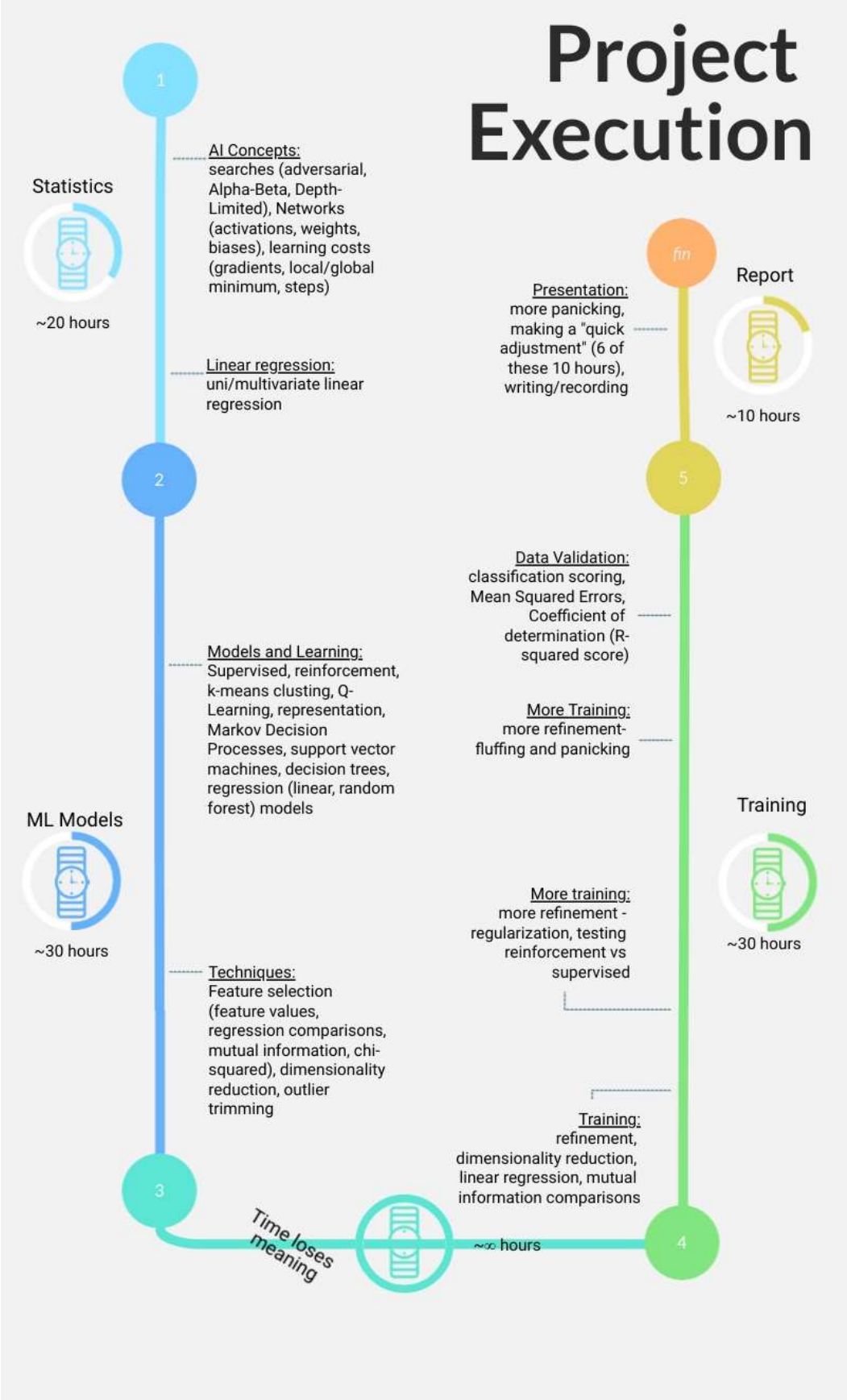
Intro to Machine Learning in Python

This project was an introduction to the models and methods used for Machine Learning projects, focused exclusively on the libraries available in Python. At the start of this project, I had hoped to create a recommendation system and thought this would be achievable through an implementation of a few different models and either extrapolating from users' previous ratings or performing a sentiment analysis of their engagement with other games. Alternatively, I would build a series of user-type profiles that could be used to bin together genre preferences and then predict which other games they would enjoy based on that. In reality, nothing beyond making a couple predictive models were finished. The 'project plan' that had been laid out was adhered to only is the vaguest of ways and the track to its current state could be described as 'meandering' and 'directionless'. Rumsfeld put it best when he said it was the "unknown unknowns" that tend to cause the most difficulty, as it certainly was with this project. Once again, I did not spend nearly as much time covering statistical knowledge as I had hoped to and I spent much more time focusing on the practical implementations of various techniques (meaning libraries and their methods) in python. On the next page is an approximation of how time was spent on the project, with a following summary of all the steps that were taken throughout the quarter. And again, the project diverted quite significantly from the original plan, and as such so did the expected user experience for the program.

To that end, there is no real "user experience" for this project, but I have attempted to comment the code well enough to be reviewed in any Jupyter environment. These comments are meant to act in part as instructions on how to adjust the code if there are changes one wishes to make to the predictive models produced or the data being used. As just navigating and describing how each portion of the code performs would take this entire essay, I will let the walkthrough video and comments within the code speak to that (pdf formats of each model walkthrough have been appended to this essay). A general order of operations for each model construction, however, is as follows; the desired data is imported and filtered with a 'test' dataframe to perform extensive manipulations on to prepare it for processing as a 'tree' frame, then the 'tree' frame is split into testing and training samples for the model, which is finally compared against the testing samples. Originally, I had envisioned the user being able to provide a URL to their BoardGameGeek profile, which the program would then scrape for data to bin the user into an appropriate category to select games for. For further customization, without needing to add data to BoardGameGeek, I imagined users would be able to select from a list of desired categories, mechanics, preferred number of players, etc. to get a customized recommendation depending on the scenario they wished to find a game for.

After the user had made these selections and/or provided their BoardGameGeek profile URL, they would be presented with a ranked list of games to choose from, where they could then see additional detail about the game or follow a link to the BoardGameGeek page for it. While it would be ideal to eventually reach a point where these features are available, there is an astounding amount of work to be done in order to implement them in any meaningful capacity. As I believe I've expressed before at other points during this project, I am beginning to understand why people go to school for nearly a decade in order to become professionals in this topic. The conclusion details all of the (currently known) steps that will be taken in the future to work towards this goal, but they are not broken down into detailed sub-tasks and will likely require long periods of research in their own right


Project Execution
1
Statistics
~20 hours
AI Concepts: searches (adversarial, Alpha-Beta, Depth-Limited), Networks (activations, weights, biases), learning costs (gradients, local/global minimum, steps)
Linear regression: uni/multivariate linear regression
2
ML Models
~30 hours
Models and Learning: Supervised, reinforcement, k-means clusting, Q-Learning, representation, Markov Decision Processes, support vector machines, decision trees, regression (linear, random forest) models
Techniques: Feature selection (feature values, regression comparisons, mutual information, chi-squared), dimensionality reduction, outlier trimming
3
Time loses meaning
~∞ hours
4
Training
~30 hours
Training: refinement, dimensionality reduction, linear regression, mutual information comparisons
More training: more refinement - regularization, testing reinforcement vs supervised
More Training: more refinement-fluffing and panicking
Data Validation: classification scoring, Mean Squared Errors, Coefficient of determination (R-squared score)
5
Report
~10 hours
Presentation: more panicking, making a "quick adjustment" (6 of these 10 hours), writing/recording
fin

To begin, this project was built off of the work done in the first half of this 406 project – Data Visualization and Analysis in Python. These were both 2-credit courses (as opposed to the usual 4), and as such I had planned to allocate only about 10 hours per week for the work to be done. This was my first mistake; to complete even the limited models being submitted I required at least twice as much time – and this is considering that I abandoned my original path which would have focused much more heavily on statistics as well as a broader understanding of Machine Learning. The course began smoothly enough, following Andrew Ng's ML course available on Coursera. I first fell behind here, when I took two weeks attempting to understand the mathematical implementations of a multivariate gradient descent model instead of one.

$$\text{repeat until convergence: } \{$$
$$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x_j^{(i)} \qquad \text{for j} := 0...\text{n}$$

Figure 1: the math that beat me

At this point, I decided it would be better to transition to a guided course that focused on practical implementations of models within Python, with the hopes of finding something that would still allow me to achieve my goals of implementing a recommendation system, even if I did not fully understand what was going on underneath the hood. I settled on consuming CS50's "Intro to AI with Python" course for a high-level understanding of concepts and edX's "ML with Python: Practical Intro" to get me familiar with basic implementations. Ultimately, I also found "Machine Learning Mastery" as an invaluable resource which contributed to both my fundamental understanding and knowledge of practical applications of various models and techniques commonly used in training models. Due to lack of time, I was not able to keep up with the two other courses.

One of the first hurdles I ran into with my project was something that I had not even realized would be an issue; data types. Prior to this moment I had never thought deeply about how mathematical models approached various data sets and envisioned them all as simply points on various graphs. This was in part due to the fact that every representation of these models I had come across never depicted anything more than a 3-dimensional system through which the model would navigate to correctly classify/predict values. I quickly learned this is entirely due to our own limitations on mentally processing anything beyond 3 to 4 dimensions and was not even remotely representative of the world in which these models operated, but more on that later. First, I had to learn about something so fundamental that I had stopped thinking about it since my first introduction to coding and memory allocation. Here, Machine Learning Mastery provided a surprisingly useful reference for constantly reminding myself why my models kept yelling at me when they could not understand the data.

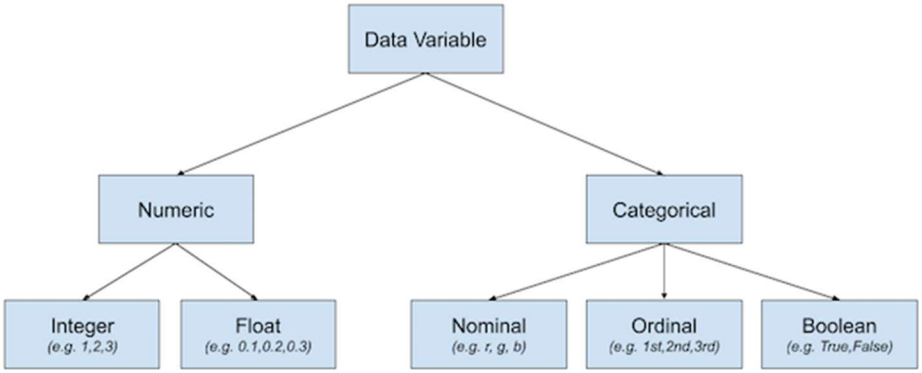


Figure 2:broken down by MLM. Be your own boss.

After an extensive amount of research to understand what each of these variable types meant to various models and prediction methods, as well as how their accuracies were checked, the system I ended up using to solve this issue was with “dummy variables” using Panda’s “get_dummies” method applied to data frames. Since my categorical variables (‘categories’, ‘mechanics’, and ‘game type’) were not ordinal, meaning one could not logically “rank” them in a series, this seemed the best way to approach the problem even though it expanded the dataframe size considerably. This method is nearly identical to the commonly

Pet
Cat
Dog
Turtle

Cat	Dog	Turtle
1	0	0
0	1	0
0	0	1

Figure 4: one-hotfix encoding

Pet
Cat
Dog
Turtle

Cat	Dog
1	0
0	1
0	0

Figure 3: dummy variable encoding

practiced “one hotfix” (used almost interchangeably from what I have seen), which takes each unique categorical value and assigns it to a Boolean value in a new column to specifically represents that value. The method of ‘dummy’ simply drops one of the columns to help simplify the dataframe for processing, as that value can be implied by having “False” values in every other ‘dummy’ column created.

Another issue I immediately encountered after selecting this method was that my data entries (games) could belong to multiple categories and have multiple mechanics. None of the model or methods I found were capable of handling cell values with any depth, so I had to create a way to expand the data set to represent all of these unique values without ruining the data set. During this process I found how heavily frowned upon it is to ever iterate through a data frame because of its extreme inefficiency compared to using the systems Pandas includes with its library, which is excellent knowledge to have once I start working with dataframes that burgeon into having hundreds of thousands of entries. While this solution did help me train the models, it went through an extensive process of testing iterations and resulted in a number of concerns with the integrity of the predictions (some of which still need to be addressed).

Once these issues with the categorical features were resolved, I began exploring implementations of the simplest classification models I could find to start wrapping my head around how these models were performing. K-Means Clustering seemed to be one of the most widely used systems for this as it also addressed creating the number of classes that could potentially be appropriate to assign all of the data within. This is done using the ‘elbow method’; first calculate the ‘inertia’ for a series of potential clusters with the number of points inside each sum of squares, then graph the series and selecting a point among the most dramatic change in slope (seen as a kink, or ‘elbow’) as the number of classes to target. Once selected, the KMeans

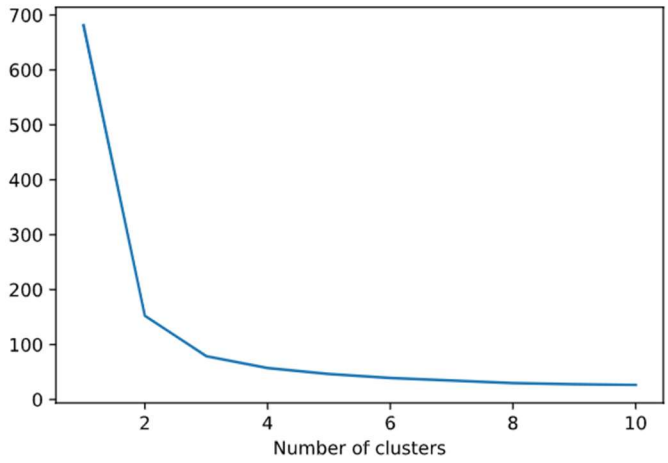


Figure 5: the 'elbow' method

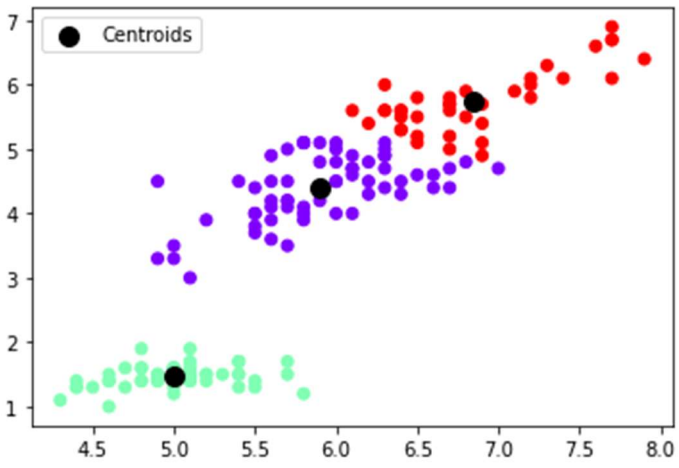


Figure 6: flattened representation of clustering

model can be fit to an adjusted dataset and determines classification based on proximity to a class centroid. This was the point at which I was faced with the harsh reality that Math has absolutely no regard for our limited perceptions of reality and expands into Nth-dimensional space far more often than science fiction has led me to believe. The 'proximity' to a centroid for my simple testing data set was only using 4 dimensions, yet I could still barely envision how it was determining whatever the closest "point" was with regard to centroids. Nonetheless, when projected into a 2D plane for graphical representation a fairly straightforward and accurate image was made. If I were to use significantly higher-dimensional data (like the boardgames with many more different classes and features) I imagine it would be far less accurate of a depiction.

Fortunately, my next approach (and one I chose to implement in the end) was much easier for my untrained meat brain to understand. Before even beginning to research Machine Learning I had heard about Data Trees as one of the most widely used systems for solving every type of learning issue. They are astoundingly simple in concept, yet can be immensely deep in their method of application. So much so that despite my focus on this singular topic for the majority of the project that I am still learning new things about them every time I re-address issues in my model. Basically, the data set begins as a single 'node' given to the model, which it then splits based on a conditional value into other nodes. At each of these new nodes, there is a new conditional evaluation and further splitting. This process is repeated until an established end-condition is met and each of the ending nodes ('leaves') represent a desired data target. *[see fig 7 for an example]* This was the system which first brought to my attention the incompatibility of some variable types and models. However, it did seem like there were solutions in other languages (Specifically, R) that are inherently capable of evaluating these relationships for splitting without the need of creating a series of pseudo-boolean 'dummy' variables.

After this approach was applied I encountered my first troubles with my method of splitting the multiple-value cells into multiple rows. My model accuracy was abysmally low (around 13%, if I remember correctly) and it took me some time to realize why. Because I had duplicated the other data in a row when creating a new row for unique categorical values, the model was being given thousands of instances of identical data which could have multiple 'true' results. However, it was only being tested against a single result and thus was 'missing' many targets that it was in reality getting correct. Once I created a custom accuracy check that would take into account all of the possible 'true' values, my accuracy score shot above 90%.

Following a series of subsequent analyses for various hyperparameter adjustments to find their ideal settings, I was able to attain a 94% accuracy rating. The first parameter analyzed was the impact of depth (how many levels and number of splits would a tree be allowed).

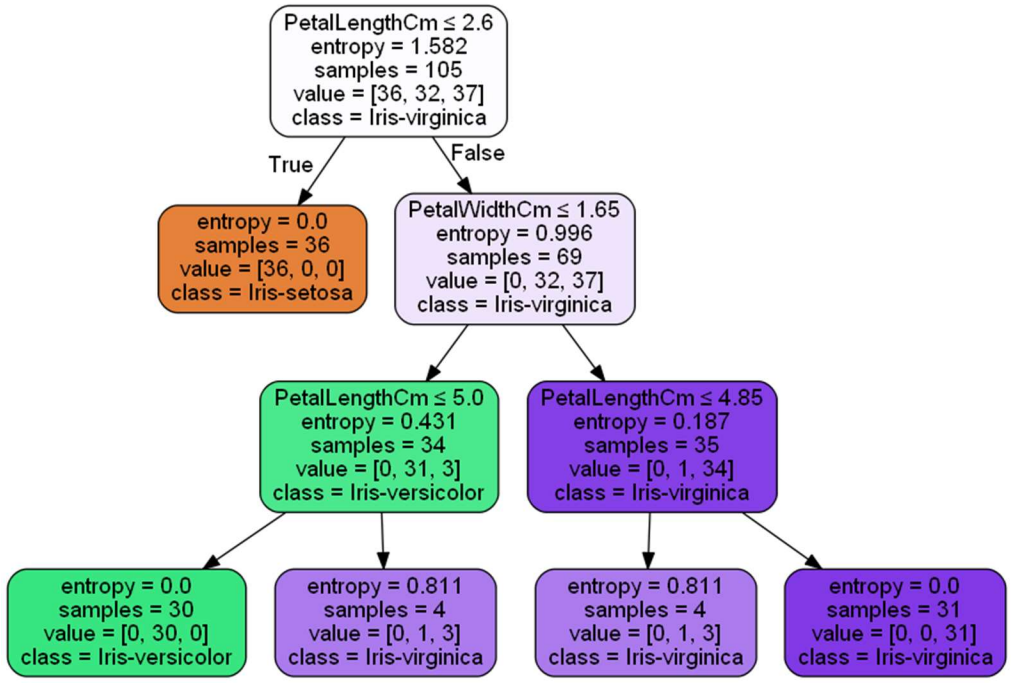


Figure 7: example of how a decision tree splits by nodes to classify data

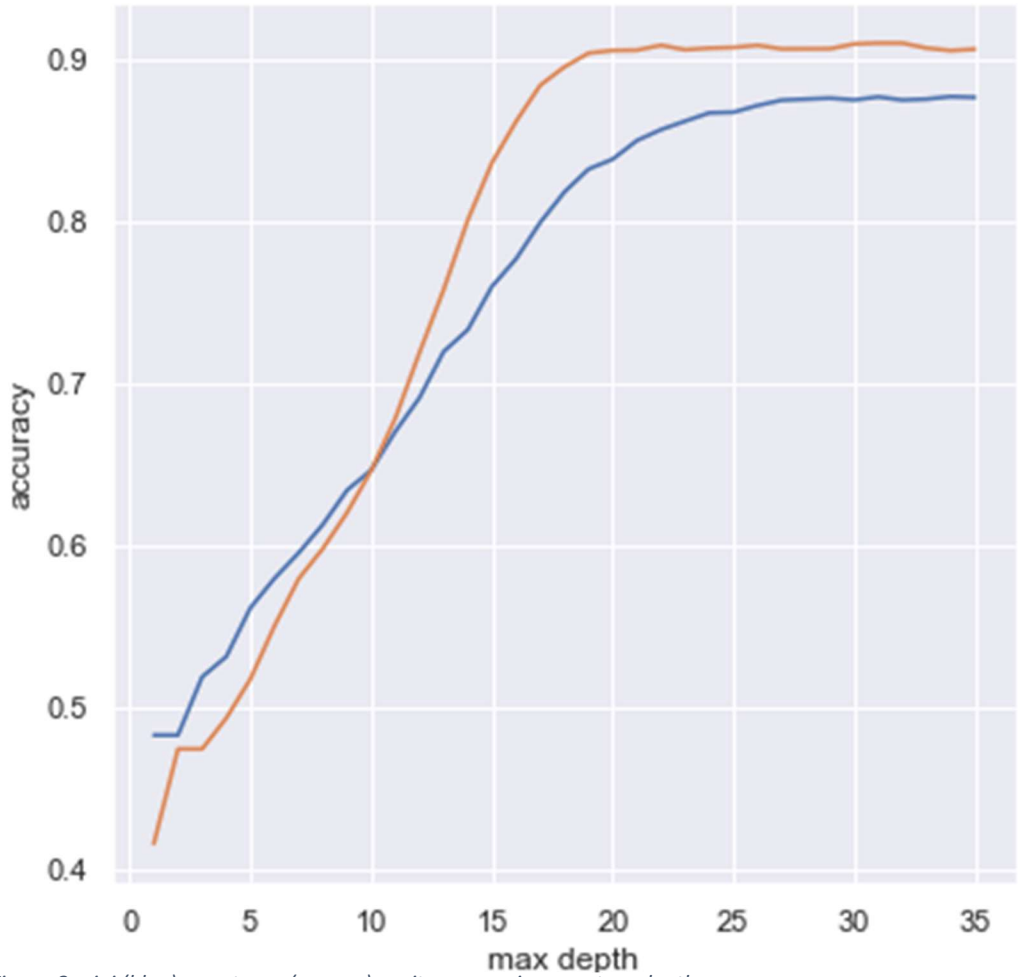


Figure 8: gini (blue) vs entropy (orange) purity accuracies over tree depth

If a model was allowed to go too deep it would not only take significantly longer to train to find the ideal model, it creates higher potential for the model to over-fit and become too specific to the exact dataset provided for training. The 'purity' assessments of entropy and gini each measure the number of unique values at a given node, which can be used to measure information gain between splits (how much more 'pure' the new nodes are). While their values are represented on similar curves, their calculations are slightly different and can lead to different splitting methods. *[see fig 8 for an example]* There is also a 'splitting type' value that can be random or 'best', 'best' being asserted by a single regression analysis of features at each node. The 'random' splits showed some surprising jumps in accuracy at lower tree depths and thus have the potential to be useful for large-scale systems, but in turn may require more iterations for training to find those ideal results. With these adjustments I was able to get the decision tree model up to a 92% accuracy.

Inspired by a community bias for newer games discovered during the data exploration half of this project, I discovered that I was able to further improve accuracy of both the regression model and decision tree model by limiting the years analyzed to the more recent decades, gaining around another 2% in accuracy. I had thought this maybe have been due to outdated categories or mechanics that were no longer appearing in modern games and thus diluting the testing pool with irrelevant prediction options, but this was not the case. After further exploration of pair plots and regression analyses I could not find a reason for why this was the case. This may be (almost certainly) due to my lack of experience with the various tools and systems at play, but may also be due to an untracked feature (eg. 'aesthetics' or 'countries sold in'). At the moment, I can only presume that older games were more homogenous in their other feature values which results in the model having a harder time with discretely classifying them.

While I was happy to achieve a 94% accuracy rating, this turned out to be an erroneous result of my solution for catching 'missed' targets; I had duplicated the number of targets to match the number of rows in the training set. While the model was *technically* correct (the best kind of correct) in its predictions, the figure was artificially inflated by giving it a number of correct/incorrect chances equal to the number of categories a target belonged to, which obviously skewed towards a higher "correct" percentage since it was specifically trained to do that. In a moment of self-congratulatory brilliance I realized the testing portion of the data had no need to be expanded since the inputs were blind to the target anyways, allowing me to significantly reduced the number of rows with duplicated data (and thus number of chances to get a category right or wrong). Currently the best model sits around 87% accuracy, but I still have numerous changes to make for feature selections and testing frame corrections.

In regard to feature selection, this turned out to be extremely important for the regression model. In approaching the various techniques for going about feature selection, I relied most heavily on ScikitLearn's extensive documentation and practiced a number of simpler applications before attempting to apply anything to my boardgame datasets. Due to my undirected approach, I unfortunately wasted a lot of time attempting to figure out which

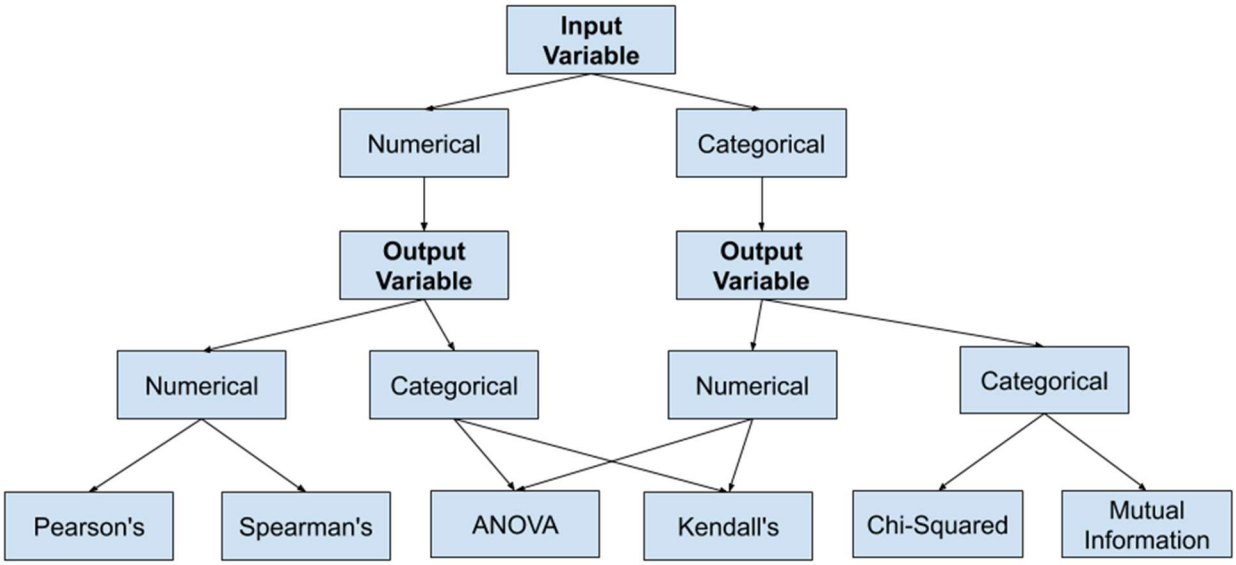


Figure 7: Feature selection based on variable values, as dictated by MLM

methods and systems of feature scoring were best for my particular use case. After too much time spent googling terminology and looking for examples, I found a lovely breakdown of which method to use based on the input and output variables that was provided by Machine Learning Mastery. These are all considered to be "filtration" methods; attempting to rank feature importance based on various relationships and then select those for the model. Machine Learning Mastery provides another breakdown for other types of feature selection which I have yet to implement in addition to the number of methods in *fig. 8* that I never found time to practice. Due to limitations on both time and my system hardware, I was able to implement only Pearson's Correlation Coefficient with the f_regression method in ScikitLearn.

I was also very interested in attempting to root out any mutual information that would only be visible when comparing multiple features to one another in higher dimensions, which is achievable though the mutual_information methods also available in SciKitLearn. I was able to successfully analyze its output in the context of a limited dataframe and see why both f_regression and mutual information methods would be helpful to generating an accurate model. Even without this, I was able to get a regression accuracy rating of around 94%.

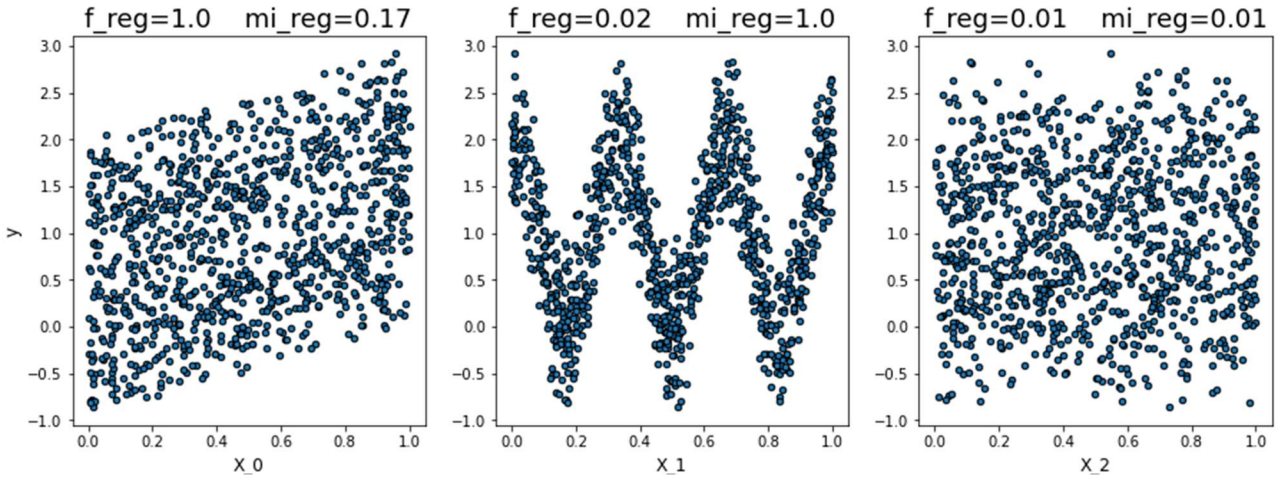


Figure 8: f_regression and mutual information (mi) scores for the 3 features in y = x0A+x1B+x2C, with feature value vs output graphs

Unfortunately, no matter how the mutual information methods were approached, they always failed due to memory issues. While observing the process I noticed that Python never consumed more than half of my available working memory and almost didn't touch the CPU or GPU at all. During the night of writing this essay I thought I made a reasonable decision to allocate a higher priority to Python running on my system (actually made it the highest priority…) and allow it to use any and all available RAM. While not especially beefy, I thought my 16GB RAM and i7-108x would be adequate since I had trimmed the dataframe (through f_regression) to only 20 variables. I was very, very wrong. As it spun up I watched the resources being consumed to see if the adjustments I had made were working. They were, and I watched in amusement as the usage climbed to nearly 100%. At its peak I took a screenshot to send to my friend, and just as I hit the "send" button in Discord my computer began to halter. Then it stopped. Then it beeped in a jarringly loud series of hauntingly familiar tones… BIOS beep

Name	99% Memory	54% CPU	99% Disk	0% Network	20% GPU
Visual Studio Code (32)	7,395.2 MB	28.8%	26.4 MB/s	0 Mbps	12.9%
Python	7,181.6 MB	21.5%	6.5 MB/s	0 Mbps	0%

Figure 9: moments before disaster

codes. I panicked and sat there for a moment hoping it would resolve itself before the screen went black. Long story short, it did not resolve itself and I had lost the entire weekend's worth of work. Fortunately, so much of the project was research and implementation that I was able to quickly implement the improvements I had made in just around 3 hours. Unfortunately, on top of not having an even remotely completed project for this quarter (although it was certainly a bit too ambitious for having only about 10 hours per week to work on it) the final report became more of a 12-hour sleep-deprived stream-of-consciousness exercise. Hopefully it was at least entertaining to read because I obviously do not have the time for more editing that I've already put in.

So, since they were an immediate need stemming from this project, I would include Windows' system recovery tools and memory diagnostics as resources that contributed to the final product. As far as actual incorporations of software I almost exclusively used packages belonging to the Python libraries of SciKitLearn (for models, metrics, and methods), Pandas (for dataframes, file handling [along with joblib], data filtering, and exploratory analysis), numpy (statistical exploration and analyses) and matplotlib (for data visualization). I also used Jupyter in the VSCode IDE exclusively to do my coding and analysis as I fell in love with the way it streamlined the entire process of importing and manipulating data for analysis or training. It was immensely helpful to be able to run cells independently and out of order, even if there were a few times I was thrown off by the persistent global variable tracking. Of course, none of this would have been possible if the folks at BoardGameGeek did not provide a wonderful API which I could use to scrape data in the first place, nor the work of my prior self to construct a system to retrieve, clean, and organize all of the data before importing it into this currently project. Thanks, past me.

As for future me, he is going to be really upset with how much work there is left to do on this project. I had intended to go into this winter break having at least the recommendation system in place so that I could build an user interface around it, but that is obviously not

happening. In all seriousness, it is a very mixed set of emotions coming to the conclusion of this quarter. I thought I had walked into the first half of this project unprepared, only to have been completely blindsided by the sheer depth of nearly *every single* part of this project. Far too much time was spent on terminology alone, just to better comprehend examples that once I understood I realized were not applicable to my goals. Obviously I am upset with myself for my habit of always wanting to change "just one small thing" leading to the disaster of memory corruption last night, but in these last panicked hours I've come to feel like I've actually learned a lot instead of being overwhelmed by the mountain of work that lays ahead of me. Working on this project has already given me the vocabulary to discuss all of the topics covered here with professional engineers and analysts on the AI team at work (T-Mobile). Apparently my enthusiasm for the work is infectious and they have recommended me to their manager for an internship position starting in January. I think my best hope to make that work out is to keep working on this project extensively in the coming weeks, which I am truly eager to do despite all of the setbacks. On that note…

Project Continuance

There is a lot of work to be done in exploring all the available ways to perform feature selection, and even with what happened I still feel tantalizingly close to being able to implement a mutual info analysis. However, I would like to first try Recursive Feature Elimination as it seems to be a simpler method to implement (and I couldn't possibly be wrong about that).

There is also an ensemble method (using numerous other methods/iterations to produce a unified output) of selection that I want to look into after how astoundingly powerful that proved to be for the regression model.

There is also a 'gradient boosting' approach which I still know next to nothing about, other than seeing others speak of it in hushed tones. It looked much more complicated to implement, but could prove extremely useful if it can be used in more contexts.

I also need to finish optimizing both the data sets and model parameters used in training. For instance, I would like to attempt compressing the expanded 'mechanics' rows with dummy variables into a single row to represent their original game again.

Only at the very end did I learn of the power of an 'ensemble' of models, creating a forest of regression model trees that all work together to increase accuracy by a staggering amount. This method is very deep and has a lot of customization, and I would like to look into making it as efficient as possible since its models are also considerably larger.

I am still familiarizing myself with the various methods of even testing model accuracy, much less building the models themselves, and need to better understand all of the alternatives before being confident that I've chosen the best ones for my use cases.

I believe the last step (before the interface) will be learning about Matrix Factorization. I believe this is currently the strongest way to build a recommendation system, or at least it is specifically designed for that purpose. I have no idea what it entails, but I'm sure it won't be something easy to implement.

Finally, after all of that, I'll just built an interface! Simple.

Game Classification Model

execute in same directory as 'game_data.csv'

In [1]:
```
import os, math
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import train_test_split
from sklearn import metrics

# needs to be reset if a model is saved
# os.chdir(working_dir_path)
working_dir = os.getcwd()
game_data = pd.read_csv('game_data.csv')
game_data.head() # See the first 5 rows to check data import
```

Out[1]:

	id	type	name	year	minplayers	maxplayers	playingtime	minplaytime	maxplayti
0	13	boardgame	Catan	1995	3	4	120	60	1
1	822	boardgame	Carcassonne	2000	2	5	45	30	
2	30549	boardgame	Pandemic	2008	2	4	45	45	
3	68448	boardgame	7 Wonders	2010	2	7	30	30	
4	36218	boardgame	Dominion	2008	2	4	30	30	

5 rows × 22 columns

clean and filter data

selective filtering for removing extremes, retain 'game_data' in case any reference to complete set is needed

In [2]:
```
bgg_games = game_data[game_data['maxplayers'] <= 30]
```

```
bgg_games = bgg_games[bgg_games['minplaytime'] <= 180]
bgg_games = bgg_games[bgg_games['maxplaytime'] <= 720]
bgg_games = bgg_games[bgg_games['minage'] <= 21]
bgg_games = bgg_games[bgg_games['playingtime'] >= 10]
bgg_games = bgg_games[bgg_games['maxplayers'] >= bgg_games['minplayers']]
```

first selection of columns potentially relevant to 'categories'

In [3]:
```
dtc_test = bgg_games[['type', 'year', 'minplayers', 'maxplayers', 'playingtime',
       'minplaytime', 'maxplaytime', 'minage', 'users_rated', 'avg_rating',
       'bay_rating', 'owners', 'traders', 'wanters', 'wishers',
       'total_comments', 'total_weights', 'complexity', 'categories',
       'mechanics']]
```

convert strings of multiple values into lists

In [4]:
```
dtc_test['categories'] = dtc_test['categories'].apply(lambda x: x.strip(']['').split(',
dtc_test['mechanics'] = dtc_test['mechanics'].apply(lambda x: x.strip(']['').split(', ')
```

count number of mechanics and categories for each game, make new columns

In [5]:
```
dtc_test['num_mechs'] = dtc_test.apply(lambda row: len(row['mechanics']), axis=1)
dtc_test['num_cats'] = dtc_test.apply(lambda row: len(row['categories']), axis=1)
dtc_test['rating_diff'] = dtc_test.avg_rating - dtc_test.bay_rating
dtc_test['player_diff'] = dtc_test.maxplayers - dtc_test.minplayers
```

In [6]:
```
def split_data_frame_list(df, target_column, output_type=str):
    '''
    Accepts a column with list values and splits into several rows.

    df: dataframe to split
    target_column: the column containing the values to split
    output_type: type of all outputs
    returns: a dataframe with each entry for the target column separated, with each ele
    The values in the other columns are duplicated across the newly divided rows.
    '''
    row_accumulator = []

    def split_list_to_rows(row):
        split_row = row[target_column]
        if isinstance(split_row, list):
          for s in split_row:
              new_row = row.to_dict()
              new_row[target_column] = output_type(s)
              row_accumulator.append(new_row)
        else:
          new_row = row.to_dict()
          new_row[target_column] = output_type(split_row)
          row_accumulator.append(new_row)

    df.apply(split_list_to_rows, axis=1)
    new_df = pd.DataFrame(row_accumulator)

    return new_df
```

split 'mechanics' lists into multiple rows for decision tree training

In [7]:
```
dtc_test = split_data_frame_list(dtc_test, 'mechanics')
```

Before filtering by year for model, split and test to remove old mechanics/categories that are no longer relevant

In [8]:
```
# year by which to remove uniques to prevent models from being incompatible
# -math.inf will include all games in dataframe
break_year = 2000 #-math.inf

old_mechs = list(dtc_test[dtc_test['year'] < break_year].mechanics.unique())
new_mechs = list(dtc_test[dtc_test['year'] >= break_year].mechanics.unique())
unique_old_mechs = list(set(old_mechs).difference(new_mechs))

# remove rows with irrelevant mechanics
dtc_test = dtc_test[~dtc_test.mechanics.isin(unique_old_mechs)]
```

second selection of columns to further filter for decision tree model

In [9]:
```
# columns to use
desired_cols = ['type', 'year', 'minplayers','maxplayers', 'playingtime', 'minplaytime'

# columns available
#all_cols = ['type', 'year', 'minplayers', 'maxplayers', 'playingtime', 'minplaytime',

# make dummies and attach to frame for tree model, leave categories alone
total_frame = dtc_test[desired_cols]
mech_dummies = pd.get_dummies(total_frame['mechanics'], prefix='mech', drop_first=True)
total_frame = pd.concat([total_frame, mech_dummies], axis=1)
type_dummies = pd.get_dummies(total_frame['type'], prefix='type', drop_first=True)
total_frame = pd.concat([total_frame, type_dummies], axis=1)
```

In [10]:
```
# preserve 'total frame' for testing with all years later
tree_frame = total_frame[total_frame['year'] >= break_year]
```

filter out categorical columns for tree fitting

In [11]:
```
# list of just desired features, now including one-hot cols and remove categorical cols
features = list(tree_frame.columns)

# remove categorical cols
features.remove('mechanics')
features.remove('type')
```

Divide the data set

split data into training portions

In [12]:
```
# filtered features with one-hot fixes for categorical columns
X = tree_frame[features]

# target variable
y = tree_frame[['categories']]

# Split method, 0.3 == 30% of data saved for testing, choosen randomly from set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=4
```

split training data so model can "learn" different categories separately

keep testing data intact for checking accuracy

In [13]:
```
# split training data by category
X_train = split_data_frame_list(X_train, 'categories')
y_train = split_data_frame_list(y_train, 'categories')

# remove target column from X_data
# typically done before, but was needed for splitting training data
X_train.drop('categories', inplace=True, axis=1)
X_test.drop('categories', inplace=True, axis=1)
```

Train the model

adjust variables prior to loop for hyperparameter adjustments

In [14]:
```
from copy import deepcopy

best_accuracy = 0
models_to_compare = 3
tree_depth = 20
for j in range(models_to_compare):
    # Decision Tree classifer object
    dtc = DecisionTreeClassifier(criterion="entropy", splitter='best', max_depth=tree_d

    # Train Decision Tree Classifer
    dtc = dtc.fit(X_train,y_train)

    # predictions by model for y
    y_pred = dtc.predict(X_test)

    # custom accuracy check
    correct = 0
    y_targets = y_test["categories"].tolist()
    for i in range(len(y_pred)):
        if y_pred[i] in y_targets[i]:
            correct += 1

    accuracy = correct / len(y_pred)
    if accuracy > best_accuracy:
        best_dtc = deepcopy(dtc)
        best_accuracy = accuracy
        print("best accuracy: ", accuracy)

print('done!')
```

```
best accuracy:  0.8910792883268241
best accuracy:  0.891513235385283
done!
```

Save model created in loop above

In [18]:
```
from joblib import dump, load

# swtich to model directory
model_dir = working_dir + "\models"
os.chdir(model_dir)

# create and save file
# [model type]_[deatils]_[accuracy]
joblib_file = "dtc_test_8752.joblib"
dump(best_dtc, joblib_file)
```

Out[18]: ['dtc_test_8752.joblib']

load and test model

In [19]:
```
# swtich to model directory and load
model_dir = working_dir + "\models"
os.chdir(model_dir)

# name of model in folder
joblib_file = "dtc_test_8752.joblib"
joblib_model = load(joblib_file)
```

get new set not limited by year (if desired for 'complete' check)

In [20]:
```
# before year trimmed
X = total_frame[features]
y = total_frame[['categories']]
```

split data into new set to compare against

In [21]:
```
# percent of data to use for test
pod = 50

# split into new set to test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=(pod/100), random_s

# then remove category column
X_test.drop('categories', inplace=True, axis=1)
```

test loaded model's accuracy

In [22]:
```
y_pred = joblib_model.predict(X_test)

correct = 0
y_targets = y_test["categories"].tolist()
for i in range(len(y_pred)):
    if y_pred[i] in y_targets[i]:
        correct += 1

print("Accuracy:", correct / len(y_pred))
```

```
Accuracy: 0.9615815233037971
```

Score Prediction Regression Model

In [1]:
```
import os
import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import PolynomialFeatures

working_dir = os.getcwd()
game_data = pd.read_csv('game_data.csv')
game_data.head() # See the first 5 rows to check data import
```

Out[1]:

	id	type	name	year	minplayers	maxplayers	playingtime	minplaytime	maxplayti
0	13	boardgame	Catan	1995	3	4	120	60	1
1	822	boardgame	Carcassonne	2000	2	5	45	30	
2	30549	boardgame	Pandemic	2008	2	4	45	45	
3	68448	boardgame	7 Wonders	2010	2	7	30	30	
4	36218	boardgame	Dominion	2008	2	4	30	30	

5 rows × 22 columns

clean and filter data

selective filtering for removing extremes, retain 'game_data' in case any reference to complete set is needed

In [2]:
```
bgg_games = game_data[game_data['year'] > 1980]
bgg_games = bgg_games[bgg_games['maxplayers'] <= 30]
bgg_games = bgg_games[bgg_games['minplaytime'] <= 180] # 120 - 90th percentile
bgg_games = bgg_games[bgg_games['maxplaytime'] <= 720]
bgg_games = bgg_games[bgg_games['minage'] <= 21]
```

```
bgg_games = bgg_games[bgg_games['playingtime'] >= 10]
bgg_games = bgg_games[bgg_games['maxplayers'] >= bgg_games['minplayers']]
```

select cells potentially relevant to rating (before community interaction)

In [3]:
```
dtc_test = bgg_games[['type', 'minplayers', 'maxplayers', 'playingtime',
       'minplaytime', 'maxplaytime', 'minage', 'avg_rating', 'mechanics',
       'bay_rating', 'total_comments', 'total_weights', 'complexity', 'categories']]
```

convert mechanics and categories into lists with values

In [4]:
```
dtc_test['categories'] = dtc_test['categories'].apply(lambda x: x.strip(']['').split(',
dtc_test['mechanics'] = dtc_test['mechanics'].apply(lambda x: x.strip(']['').split(', ')
```

count number of mechanics and categories for each game, make new columns

In [5]:
```
dtc_test['num_mechs'] = dtc_test.apply(lambda row: len(row['mechanics']), axis=1)
dtc_test['num_cats'] = dtc_test.apply(lambda row: len(row['categories']), axis=1)
dtc_test['player_diff'] = dtc_test.maxplayers - dtc_test.minplayers
```

In [6]:
```
def split_data_frame_list(df, target_column, output_type=str):
    '''
    Accepts a column with list values and splits into several rows.

    df: dataframe to split
    target_column: the column containing the values to split
    output_type: type of all outputs
    returns: a dataframe with each entry for the target column separated, with each ele
    The values in the other columns are duplicated across the newly divided rows.
    '''
    row_accumulator = []

    def split_list_to_rows(row):
        split_row = row[target_column]
        if isinstance(split_row, list):
          for s in split_row:
              new_row = row.to_dict()
              new_row[target_column] = output_type(s)
              row_accumulator.append(new_row)
        else:
          new_row = row.to_dict()
          new_row[target_column] = output_type(split_row)
          row_accumulator.append(new_row)

    df.apply(split_list_to_rows, axis=1)
    new_df = pd.DataFrame(row_accumulator)

    return new_df
```

split lists into multiple rows for regression model

In [7]:
```
dtc_test = split_data_frame_list(dtc_test, 'categories')
dtc_test = split_data_frame_list(dtc_test, 'mechanics')
```

get all desired cols and apply one-hot fix to categorical features

In [8]:
```
# for filtering cols
desired_cols = ['type', 'minplayers', 'maxplayers', 'playingtime', 'minplaytime', 'maxp
```

```
# make dummies (one-hot fix) for categorical values
# will remove categorical columns as well
model_frame = dtc_test[desired_cols]
model_frame = pd.get_dummies(model_frame, drop_first=True)
```

filter out columns for model fitting

In [9]:
```
# list of just desired features, now including one-hot cols
features = list(model_frame.columns)

# remove target cols
features.remove('avg_rating')
features.remove('bay_rating')
```

Divide the data set

split data into training and testing portions

In [10]:
```
# filtered features with one-hot fixes for categorical columns
X = model_frame[features]

# remove undesired columns
X.drop("categories_'Expansion for Base-game'", inplace=True, axis=1) # duplicate of "is

# target variable - average because bays is weighted so heavily
y = model_frame[['avg_rating']]

# Split data - limit training because it will be filtered and re-allocated later
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.0001, random_stat
```

feature selection

In [11]:
```
from sklearn.feature_selection import SelectKBest, f_regression, mutual_info_regression
from functools import partial

# f_regression is univarite - direct correlations
# mutual_info compares multiple feature pairs

def feature_selection(X_train, y_train, m=0, n='all'):
        '''produces feature values to help with selection
        IN: 3 frames split from data
                m = type of method desired (int)
                n = number of top features to select
        OUT: transformed X-data and feature selection model'''

        # partial to establish params for mutual info
        # CAN'T HANDLE
        mutual_info = partial(mutual_info_regression, random_state=0)

        # scoring functions to use
        methods = [f_regression, mutual_info]

        # configure to select type of feature value grader and number of top features t
        fs = SelectKBest(score_func=methods[m], k=n)

        # correlate relationships from training data
        fs.fit(X_train, y_train)
```

```
        return fs
```

feature selection parameters for filtering columns into regression model

In [12]:
```
# feature selection scores (currently set to check all features)
fs = feature_selection(X_train, y_train, 0, 15)
feature_mask = fs.get_support()
top_features = X.columns[feature_mask]
```

reset test data for mutual information filtering

In [13]:
```
X = model_frame[top_features]

# still the same
y = model_frame[['avg_rating']]

# new data split for training with limited set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=2
```

In [14]:
```
# # DANGER DANGER
# # new feature selection model
# fs_mut = feature_selection(X_train, y_train, 1, 15)
# feature_mask = fs.get_support()
# top_features = X.columns[feature_mask]
# print(top_features)
```

Train the model

In [17]:
```
from sklearn.linear_model import LinearRegression
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error, r2_score
from copy import deepcopy
import math

# for checking accuracy
best_r2 = 0
best_mse = math.inf

# random forest regressor creates many regression trees for evaluation
rfr = RandomForestRegressor(n_estimators=100, max_depth=20, random_state=42)

# fit model to training data
rfr.fit(X_train,y_train)

# predictions by model for y
y_pred = rfr.predict(X_test)

# accuracy check, lower is better
mse = mean_squared_error(y_test, y_pred)
print('MSE Forest: ', mse)

# The coefficient of determination: 1 is perfect prediction
r2 = r2_score(y_test, y_pred)
print('r2 Forest: ', r2)
```

```
MSE Forest:  0.050877176437359596
r2 Forest:  0.9411821686719545
```

In [18]:
```
# check specific instances for accuracy
target = 170 # must be smaller than dataframe length and >=0
print("target score: ", y_test.iloc[target])
print("predicted score: ", y_pred[target])
```

```
target score:  avg_rating    8.22286
Name: 42866, dtype: float64
predicted score:  8.218860365220493
```

save model and test accuracy of accuracy rating

In [19]:
```
from joblib import dump, load

# swtich to model directory
model_dir = working_dir + "\models"
os.chdir(model_dir)

# create and save file
# [model type]_[details]_[accuracy]
joblib_file = "rfr_test_9412.joblib"
dump(rfr, joblib_file)

# test load
joblib_model = load(joblib_file)

y_pred = joblib_model.predict(X_test)

# accuracy check
mse = mean_squared_error(y_test, y_pred)
print('MSE: ', mse)

r2 = r2_score(y_test, y_pred)
print('r2: ', r2)
```

```
MSE:  0.050877176437359596
r2:  0.9411821686719545
```

load and test model

In [20]:
```
# swtich to model directory
model_dir = working_dir + "\models"
os.chdir(model_dir)

# file name with model to load
joblib_file = "rfr_test_9412.joblib"
joblib_model = load(joblib_file)
```

re-load data to check against different set, after initial test

(needs to be run twice for some reason...)

In [23]:
```
X = model_frame[top_features]

# still the same
y = model_frame[['avg_rating']]

# new data split for training with limited set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=2
```

re-test model

In [24]:
```
# model prediction scores
y_pred = joblib_model.predict(X_test)

# accuracy check
mse = mean_squared_error(y_test, y_pred)
print('MSE: ', mse)

r2 = r2_score(y_test, y_pred)
print('r2: ', r2)
```

```
MSE:  0.039372519846675114
r2:  0.9547505971722409
```