Using Imbalanced Classification to Improve NCAA Tournament Upset Detection

THE PROBLEM

In the annual men's college basketball tournament, many games involve well-regarded "favorites" against an "underdog" team considered unlikely to win. On average, 20% of these games result in a victory for the underdog, a.k.a. an "upset". These games are troublesome for millions of bracket contest players who enter tournament prediction pools every year. Upsets are predicted with lower accuracy than other games. When strong title contenders are defeated by a weaker team in an upset, the upset also damages the scoring potential for bracket contest players who predict a deep run for the upset victim. Machine learning could provide better prediction guidance for bracket contest players and sports gamblers, but most classification algorithms are not optimized for this "unbalanced" classification problem.

The goal of the project is use multiple machine learning algorithms to classify the outcomes of tournament games between "favorites" and "underdogs", and to use unbalanced classification approaches to improve classification performance.

The Client(s)

- Over 70 million individuals who complete tournament brackets for informal group contests, aka "office pools". This group wagered an estimated $10.4 billion in 2017.
- Individuals placing bets on NCAA tournament games
- Media outlets offering data-driven tourney guidance via television or web content

THE DATA

Data for this project was obtained and processed in a previous project on NCAA tournament games. The data sources can be described under two categories, which I will refer to as "Kaggle contest data" and "Supplemental data".

Kaggle contest data

For an annual NCAA tournament prediction contest ("March Machine Learning Mania"), the data science contest website Kaggle (https://www.kaggle.com/) provides a group of .csv files. Below I list each individual file, a brief description of file contents, and examples of the type of data resulting from processing each file.

Regular season results	*Contents:* Scores and box-score statistics from pre-tournament games *Example features:* Average points per-game, rebounds per game, turnovers per-game, 3-point shooting %, free-throw shooting %
Coaches	*Contents:* Coach names with teams coached and job start/end dates *Example features:* Has won a tourney game, highest tourney round reached, number of elite-8 appearances, has coached in tourney
Conferences	*Contents:* Name of team's conference *Example features:* Team plays in a "Power" conference or not

Team and game geography	*Contents:* Latitude/longitude of school and tourney game locations *Example features:* Distance (in miles) of campus from game location
Tourney seeds	*Contents:* Team seed (a tournament ranking assigned to all participating teams) for each year of tournament participation *Example features:* Team seed, opponent seed difference
Tourney results	*Contents:* Participating teams and scores for all previous NCAA tourney games *Labels:* Result of game was an upset (1) or not (0)

Supplemental data

To create additional features believed to be important for prediction of upsets, supplemental data was obtained from additional sources. This additional data was accessed via downloadable .csv files or scraping web pages.

Roster data www.sports-reference.com www.espn.com	*Contents:* Year, height, and position of each team's players *Example features:* Number of senior starters, average team height, starter average experience
Player data	*Contents:* Player-level regular-season box score statistics *Example features:* Starter/bench team scoring %, frontcourt scoring %, backcourt scoring %, guard assist/turnover ratio
Adjusted team efficiency metrics (www.kenpom.com)	*Contents:* Ratings of team offensive and defensive quality adjusted for pace-of-play, game location, and quality of opponent *Example features:* Team adjusted offensive/defensive efficiency, team adjusted efficiency margin
All-Americans www.sports-reference.com	*Contents:* List of annual All-American player award winners *Example features:* Team # of All-Americans

After data acquisition and processing, the team feature dataset contained features for 896 separate teams (64 tournament teams for each of 14 annual tournaments from 2003 to 2016).

Because each game involves a matchup of 2 teams, the tourney game outcome data was merged with the team feature data. This process resulted in a dataset where each row contained the team features for both teams involved in each of 882 NCAA tournament games (63 games x 14 tournaments) as well as the final scores for each tournament game.

Upset-Potential Matchups

To focus the analysis on games with "upset potential", I used the team seed difference to define upsets. I define a game as having "upset-potential" if it involved a team seeded at least 4 slots higher than their opponent, such as 1-seed vs. 5-seed or higher, a 2-seed vs a 6-seed or higher, and so on. All games not meeting this criterion were removed from the dataset, which reduced the dataset to 584 games (65% of games).

DATA PREPARATION

Prior to model training and testing, the tourney game scores were removed from the dataset, and the upset labels were separated from the features into a distinct label vector. Using an 80:20 ratio, the dataset was also split for training (475 examples) and testing (109 examples). The features for both the training and test sets were normalized to the same scale, with the training set used to fit the python *StandardScaler* function.

BASELINE PERFORMANCE FROM PREVIOUS RESULTS

Here I summarize my previous work on this problem, which provides as a comparative baseline for the current project. For a more detailed report on the original project, see here:

https://github.com/mworles/portfolio/blob/master/reports/upset_prediction_1_report.pdf)

Using the same dataset, I trained a logistic regression and random forest for upset classification. These models were trained on a four distinct subsets of the features to determine whether a subset of the feature groups would produce superior classification performance as compared to the full feature set. For both algorithms, I performed hyperparameter grid search with 5-fold cross-validation to identify optimal hyperparameter values for each feature set. The criterion used for model selection was highest cross-validation *F1* score. Each tuned model was then trained on the training set and used to classify upsets in the test set.

An *L2* regularized logistic regression produced the top test upset classification *F1* score of 0.49. As an implementation case, I used the trained model to classify upsets in the 2017 tournament. Performance for the 2017 tournament was *F1* = 0.60 with 90% upset recall and 45% upset precision. The *F1* score of 0.60 on the upset potential games outperformed a calculated 99% of upset *F1* scores from the 2017 Kaggle contest entries. A simulated ROI analysis also found that placing money-line bets on each 2017 tourney upset-potential game on the model-predicted winner would have achieved a 17% return.

IMPROVING ON PREVIOUS RESULTS

Given this baseline performance, **the goal of the current project is to produce a model that performs better than 0.49 upset classification F1 on the test set.** To test potential client impact, the top-performing models will also be evaluated on upset classification and ROI for the 2017 tournament.

Compared to the previous project, I use three new overall approaches to achieve this goal.

Additional algorithms. In addition to logistic regression and random forests, this project also tests support vector machines, gradient-boosted decision trees, and neural networks.

Optimized hyperparameter search. Searching over a pre-defined grid is only one way to identify optimal model hyperparameters. In this project I implement an algorithmic approach to identification of optimal model hyperparameters with *hyperopt,* an open-source Python library,

(https://github.com/hyperopt/hyperopt). In my prior results, several of the tuned models performed significantly worse on the test set than the average cross-validation score. Inspection of the grid search results revealed these models also had larger discrepancies between higher training scores and cross-validation scores, suggesting the presence of "overfitting". This project makes a concerted effort to reduce the likelihood of overfitting, by searching in a more detailed parameter space, and by attempting to identify hyperparameter values that limit overfitting.

Resampling for imbalanced classification. Classification of NCAA tournament upsets is an "imbalanced" classification problem because the upset class comprises a clear minority of the examples (78% of games are non-upsets, 22% are upsets). This class imbalance typically leads to degraded performance for most classification algorithms. One approach to address this problem is to use a "resampling" method to reduce the imbalance for model training. In this project I test the performance impact of three resampling methods.

TRAINING ON "NATURAL" LABEL DISTRIBUTION

To establish a baseline performance of each algorithm on the "natural" imbalanced label distribution, I performed model selection for each of the 5 algorithms, using an optimized hyperparameter search and 5-fold cross-validation to select the top candidates for each algorithm. Although I had prior baseline performance for logistic regression and random forests, these two algorithms were still included for hyperparameter tuning and model selection here, to test the potential performance gain from the use of *hyperopt* for hyperparameter tuning and inspection of overfitting.

Hyperparameter optimization and model selection

Here I describe my approach in using the *hyperopt* library to identify optimal hyperparameter values for the five algorithms. I used a similar general approach for each algorithm. To illustrate the process, code for only neural networks is shown below.

Orientation to the hyperopt library.
The Github README file describes `hyperopt` as a "Python library for serial and parallel optimization over awkward search spaces, which may include real-valued, discrete, and conditional dimensions". One of the core functions is `hyperopt.fmin()`, which searches over a given space to minimize an objective function defined by the user. The search space is defined using stochastic expression modules available under `hyperopt.hp`, such as `hp.randint()` for random integers within a supplied range, or `hp.choice()` for a random choice from a list of discrete options. The search algorithm has two options, `random.suggest` which searches over the space randomly, and `tpe.suggest` which uses Tree of Parzen Estimators to iteratively guide the search using results from the minimized objective function. When using `tpe.suggest`, the search adapts to the results over time by focusing the search on areas of the space that tend to produce lower values on the minimized objective function. I used `tpe.suggest` for each hyperparameter search.

Use of the hyperopt library.

For each algorithm, I wrote a script to run from the Ipython shell. Below I describe the key components of the script for neural networks.
I first set up a few objects for the *hyperopt* to use and write results. Note that the search space is defined outside the script and imported (shown in detail further below). This structure of defining the space outside the script isn't necessary, but I found this was a useful way to modify the search space and document changes during sequential runs.

```
# set number of trials and score function to use
evals = 500
scorefunc = f1_score
space_to_use = 0

# %% import search space dict from neuralnet space file
sys.path.append('C:\Users\mworley\Dropbox\capstone_two\src\models')
import neuralnet_space
space = neuralnet_space.space
space = neuralnet_space.spaces[space_to_use]

# set location for trials pickle
# trials pickle is saved throughout hyperopt run
mpath = "C:\Users\mworley\Dropbox\capstone_two\models"
place = mpath + '\neuralnet_hyperopt_0'
trials_temp = place + '_trials_temp.p'
trials_file = place + '_trials.p'

# initials trials object to save search results
trials = Trials()
```

Here I set up a code() function for the hyperopt procedure (necessary for parallel processing on my PC), beginning with importing and normalizing the training data and initializing the base model for the classifier.

```
def code():
    """Code to run hyperparameter optimization."""
    # load the data
    data_path = "C:\Users\mworley\Dropbox\capstone_two\data\processed"
    x_train = pd.read_csv(data_path + r'\xtrain_1.csv')
    xsub = x_train.drop(['season', 'rnd'], axis=1)
    y_train = pd.read_csv(data_path + r'\ytrain_1.csv', header=None)
    ytrain = y_train.iloc[:, 0]
    sc = StandardScaler()
    xtrain = sc.fit_transform(xsub)

    # set up the base model
    clf = MLPClassifier(random_state=0, solver='sgd', max_iter=2000)
```

Here I define the objective function for `hyperopt.fmin()` to minimize. In this implementation, I perform 5-fold cross-validation and obtain the mean F1 score for the 5 cross-validation folds. The objective function is written to minimize (1 – *F1*) and therefore maximize *F1* score. I also fit the classifier to the full training set and store the training set F1 score, and attach both the mean cross-validation score and the training set score to the trials object. This allows me to compare cross-validation and training scores after the conclusion of the search for inspection of possible overfitting. I also iteratively write the trials object pickle to a file, to save results in-progress in the event of an interruption or error.

```
    def objective(params):
        """Define the objective function to minimze."""
        # set classifier hyperparameters form the search space
        clf.set_params(**params)

        # perform 5-fold cross-validated and get mean score
        shuffle = StratifiedKFold(n_splits=5, shuffle=True, random_state=0)
        scores = cross_val_score(clf, xtrain, ytrain, cv=shuffle,
                                 scoring=scoretype, n_jobs=-1)
        cvscore = scores.mean()

        # get predictions and scores for full training set
        clf.fit(xtrain, ytrain)
        preds = clf.predict(xtrain)
        score_t = scorefunc(ytrain, preds)

        # attach scores to trials object and write to file
        n = len(trials.trials) - 1
        trials.trials[n]['score_valid'] = cvscore
        trials.trials[n]['score_train'] = score_t
        trials.trials[n]['score_valid_all'] = scores
        pickle.dump(trials, open(trials_temp, "wb"))

        # minimize 1 - the classification score
        return 1 - cvscore
```

The script ends with the call to run the `hyperopt.fmin()` function by passing to `fmin()` the previously defined objective function, the search space dictionary, the algorithm for the search, the number of trials, and the trials object for the search results. I also write trials object pickle to a file after completion of the search, and end the code() block function to run the code.

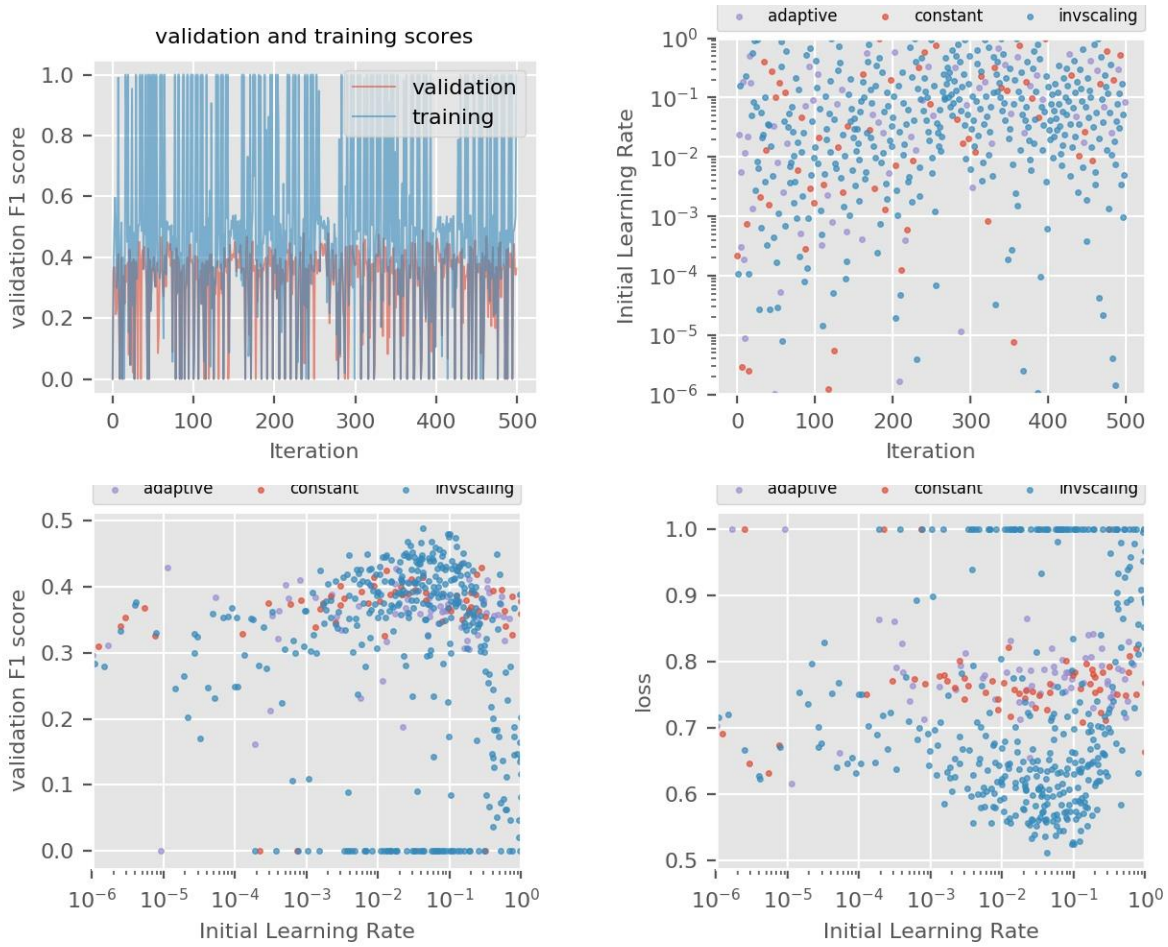
```
    # Run the hyperparameter search using the tpe algorithm
    best = fmin(objective,
                space,
                algo=tpe.suggest,
                max_evals=evals,
                trials=trials)

    # pickle and store trials object
    pickle.dump(trials, open(trials_file, "wb"))

if __name__ == '__main__':
    code()
```

As mentioned above, I wrote a script to define the hyperparameter search space outside of the main *hyperopt* script. This workflow was useful for tracking modifications to the search space in-between runs. For each algorithm, I completed sequential runs of the hyperopt script in the effort to find optimal hyperparameters and adjust the search from a "rough" to "fine", expand the search space if optimal values were near the boundary, or remove values with poor performance.

The example below for neural networks illustrates the variety of stochastic expressions available (discrete choice, log uniform, uniform, and random integer, among others).

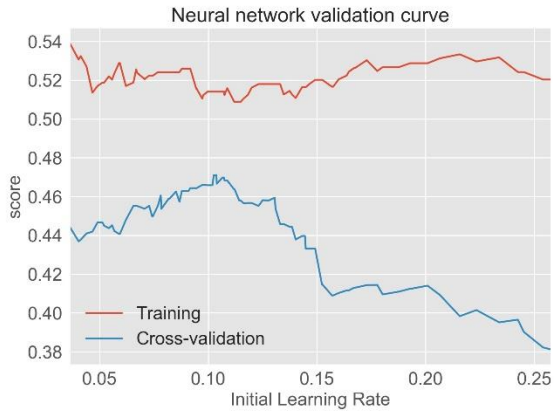
```
space_0 = {}
activations = ['logistic', 'tanh', 'relu']
space_0['activation'] = hp.choice('activation', activations)
learning_rates = ['constant', 'invscaling', 'adaptive']
space_0['learning_rate'] = hp.choice('learning_rate', learning_rates)
space_0['learning_rate_init'] = hp.loguniform('learning_rate_init',
                                              np.log(10**-6),
                                              np.log(10**0))
space_0['hidden_layer_sizes'] = 1 + hp.randint('hidden_layer_sizes', 250)
space_0['alpha'] = hp.loguniform('alpha', -9.2106, 0.00995)
space_0['momentum'] = 1 - hp.loguniform('momentum', np.log(10**-2),
                                        np.log(10**-.05))
space_0['power_t'] = hp.uniform('power_t', 0.2, 0.8)
spaces = [space_0]
```

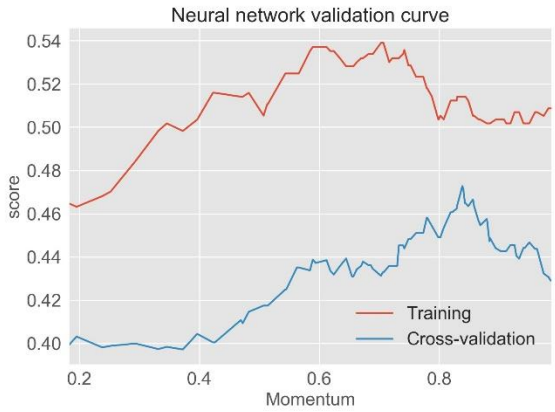
Inspection of hyperopt results to refine the search.
The goal of the hyperparameter search for each algorithm was to maximize the cross-validation F1 score, while minimizing the possibility of overfitting. For each algorithm, I began the search with a broad range of values for each hyperparameter believed to exert meaningful impact on classifier performance. Following the completion of the search, I produced plots for each hyperparameter to identify areas of the space with the highest cross-validation *F1* scores. Examples of the plots for neural networks are shown below. For each search, I plotted the mean cross-validation scores and training scores over the trials (upper left). For each hyperparameter examined in the search, I plotted the parameter values over the trials (upper right), and plotted the cross-validation score over the parameter values (lower left). I also created a 'loss' metric that accounted for both cross-validation score value and the differences between training and cross-validation scores, and plotted the 'loss' value over the parameter values (lower right).



When inspecting the plots, I look for patterns of change in the parameter values over iterations that coincide with smaller differences between cross-validation scores and training scores. I also look for ranges of the hyperparameter with higher cross-validation scores and lower loss values. In this case, the 'invscaling' learning rate schedule with initial learning rate 10^{-3} to 10^{0} achieves all of these targets. For the next search, I narrowed the initial learning rate to this range, and continued the *hyperopt* search with an additional 500 iterations on the same trials object, which allows the search to continue using the results learned during the first 500 iterations. Narrowing the search space for a given hyperparameter allowed for a finer search, and often had a "stabilizing" effect in the search for other hyperparameters. In this case, optimal values for momentum and alpha narrowed after narrowing the initial learning rate range.

After performing sequential hyperopt runs and narrowing the ranges for optimal hyperparameter values, I used the top cross-validation F1 scores to set values for all but one hyperparameter, and performed a search for single hyperparameter at a time. In this final step I used validation curves to identify the hyperparameter value that appeared to produce the best *F1* score without overfitting. Below are two examples for the neural network search.



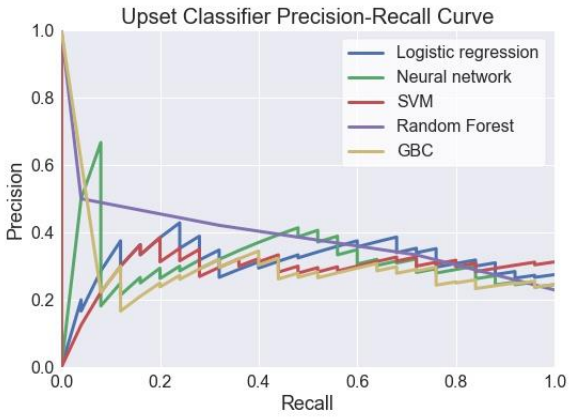


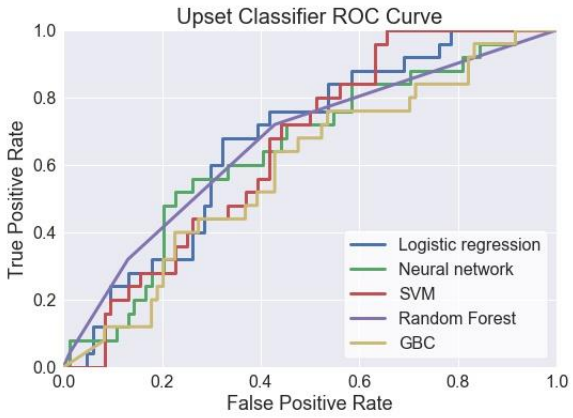
Final models for test set performance evaluation were selected after setting hyperparameters at values indicated by the individual validation curves. For hyperparameters that did not have clear optimal values from the validation curves (e.g., hidden layer size), I selected the hyperparameter value that maximized cross-validation *F1* score. Note that training scores could not be obtained for the tree-based algorithms (random forests, gradient-boosted classifier) from the *scikit-learn* implementations. In this case, hyperparameters were optimized to maximize the cross-validation *F1* score, with an effort to narrow hyperparameter values to ranges that were less "noisy" in cross-validation scores, which in theory would provide more stable estimates of generalized performance.

Test set performance

For each algorithm one model was selected for evaluation of performance on the held-out test set, which includes 109 games (23% upsets). The test set performance results are show below.

	F1	**True Positive Rate (Recall)**	**True Negative Rate (Specificity)**	**Precision**	**Precision Recall AUC**	**ROC AUC**	**Cross-Validation F1**	**Training Set F1**
Logistic regression	0.48	0.68	0.65	0.37	0.33	0.68	0.45	0.48
Neural network	0.44	0.48	0.79	0.40	0.34	0.65	0.47	0.55
Support vector machines	0.34	0.36	0.77	0.32	0.31	0.66	0.39	0.61
Random forests	0.36	0.32	0.87	0.42	0.34	0.67	0.36	NA
Gradient-boosted classifier	0.39	0.60	0.57	0.29	0.28	0.59	0.40	NA





When training on the natural imbalanced labels, the top test set *F1* score was 0.48 with Logistic Regression, very close to the prior results which had top test set *F1* of 0.49. The random forest *F1* improved from the prior results (0.36 from 0.15). The minimal reduction from the cross-validation *F1* scores to the test set *F1* scores suggests the process of hyperparameter tuning with cross-validation provided good estimates of generalized performance. Most of the algorithms achieved higher recall than precision, meaning the models are relatively better at identifying the true upsets than avoiding false positive predictions. The exception is random forests, which has higher upset precision than upset recall.

TRAINING ON RESAMPLED DATA

As stated above, most classification algorithms exhibit optimal classification performance when the dataset has imbalanced labels. But in the real-world, most classification problems involve imbalanced data, and this dataset is no different. The upsets comprise only 22% of the games. To attempt to improve the results above, I repeated the full process of hyperparameter tuning, model selection, and test set evaluation with resampled datasets.

Selection of resampling methods

For each algorithm, I tested three resampling approaches.

*Random undersampling of the majority class***.** Hereafter referred to as "undersampling", this method involves selecting random examples from the majority class, to form a dataset that includes all of the minority class examples (the "upsets" in this project), and a partial subset of the majority class examples (the "non-upsets". By reducing the size of the majority class, the labels become more balanced.

Random oversampling of the majority class. Hereafter referred to as "oversampling", this method involves selecting random examples from minority class and repeating them, to form a dataset that includes all the majority class examples, and repeats of examples from the minority class. By increasing the size of the minority class, the labels become more balanced.

Synthetic minority oversampling technique. Hereafter referred to as "SMOTE", this method involves creating new, synthetic examples that are similar in the feature space to the minority class examples. By synthetically creating new examples of minority class, the labels become more balanced.

Resampled datasets were created from the training set of 475 examples with the Python library imbalanced-learn, which has a separate function for each resampling method. By passing the function the original feature array, the label vector, and an optional ratio parameter, the function returns a resampled feature array and label vector, with the specified ratio of majority: minority examples.

For each function, the default resampling ratio is 1:1, which returns a dataset of completely balanced classes. While I considered using only the default 1:1 ratio, prior to the project I read a study showing that specifying a suboptimal resampling ratio can negatively impact classification performance, with the optimal ratio varying across algorithms and datasets. To experiment with different ratio levels, I tested four separate majority:minority ratios for each resampling method. The ratios resulting label distributions are shown in the table below. The table highlights that 12 different configurations for resampling were tested, 3 resampling methods * 4 ratios.

	3:1 ratio Not upsets: Upsets	2:1 ratio Not upsets: Upsets	4:3 ratio Not upsets: Upsets	1:1 ratio Not upsets: Upsets
Undersampling	306: 102	204: 102	136: 102	102: 102
Oversampling	373: 124	373: 186	373: 279	373: 373
SMOTE	373: 124	373: 186	373: 279	373: 373

Hyperparameter optimization and model selection

As compared to the first stage of the project where each algorithm was trained on the same training set, the decision to use test 3 resampling methods with four ratio levels created an additional layer of complexity for hyperparameter tuning and model selection. Tuning each algorithm on 12 different datasets did not seem like a very efficient plan.

Instead of tuning each algorithm separately for the 12 unique combinations of resampling type and ratio, each algorithm was tuned separately for each resampling type, and the selection of resampling ratio was included in the hyperparameter search. For each algorithm, I modified the hyperparameter optimization script and search space script to allow the hyperparameter search to include the resampling ratio as an additional object for the search. The portion of code below illustrates the initial search space for L1 logistic regression with random undersampling.

```
74  space['logisticregression__penalty'] = hp.choice('penalty', ['l1'])
75  space['logisticregression__C'] = hp.loguniform('C', np.log(0.01), np.log(100)
76  space['randomundersampler__ratio'] = hp.choice('ratio',
77                                                 [resamp_under33, resamp_under50,
78                                                  resamp_under75, 'auto'])
```

Where the list following **"`hp.choice('ratio',`"** is the list of options to pass the random undersampling function for the four different resampling ratios. The string 'auto' passes the default 1:1 ratio. The remaining objects are functions I wrote return the remaining ratios. Each function takes as input the original vector of labels, and returns a dictionary specifying the majority and minority class sizes for the resampler to return. Below is the function to return a 3:1 ratio with majority undersampling:

```
def resamp_under33(y):
    vals, counts = np.unique(y, return_counts=True)
    min_count = counts[1]
    maj_count = counts[0]
    min_label = vals[1]
    maj_label = vals[0]
    maj_count_new = min_count * 3
    return {min_label: min_count,
            maj_label: maj_count_new}
```

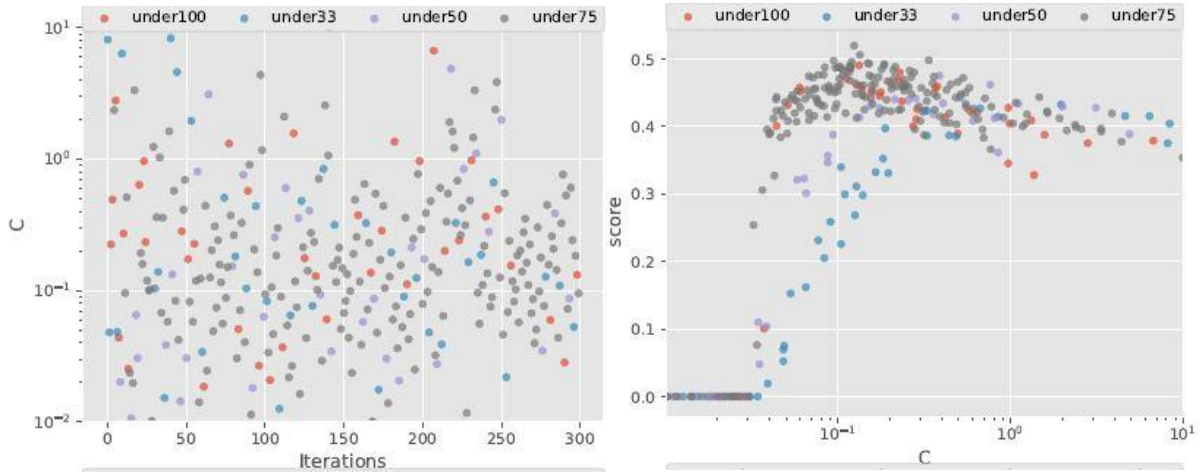
With this code now built into the search space script for each algorithm, the hyperparameter optimization script was modified to combine the base classifier and base resampling function in a pipeline object, as shown below. Also shown are the imported space object and objective function for the *hyperopt* search to minimize.

```
        # set base resampling model
        sampler = RandomUnderSampler(random_state=0)
        # set base classifier
        clf = LogisticRegression(random_state=0)
        # define pipeline
        pipe = make_pipeline(sampler, clf)
        # import the search space and select from list
        import logistic_resamp_space as space_mod
        space = space_mod.spaces[space_to_use]
        # define the objective function to minimize
        def objective(params):
            pipe.set_params(**params)
            shuffle = StratifiedKFold(n_splits=5, shuffle=True,
                                      random_state=0)
            scores = cross_val_score(pipe, xtrain, ytrain, cv=shuffle,
                                     scoring='f1', n_jobs=-1)
            cvscore = scores.mean()
            return 1 - cvscore
```

Inside the objective function, the pipeline is fit with `cross_val_score` to return the array of scores from 5-fold cross validation. Note that for each fold, resampling occurs *after* the cross-validation split. This distinction is especially important for random oversampling and SMOTE, to ensure that repeats or synthetically-generated examples of the training set do not "leak" into the validation fold.

For each *hyperopt* trial, the resampling ratio and hyperparameter values are selected from the imported search space. Structuring the project this way, in combination with the `tpe.suggest` algorithm in *hyperopt*, allowed the hyperparameter search to "learn" the optimal resampling ratio alongside the algorithm hyperparameters. As I manually narrowed the algorithm hyperparmeter search space for sequential runs of the *hyperopt* script, I also manually removed resampling ratios with the worst performance.

To illustrate this process, below I show results from the first run of the hyperparameter optimization script for *L1* logistic regression with undersampling. The plots show the *C* parameter values over the course of 300 trials (left), and the mean cross-validation *F1* scores over the C parameter values (right). Colors of scatter points show the undersampling ratio for each trial. The left plot shows a "preference" over time for the 4:3 undersampling ratio, shown by the proportional increase in grey scatter points. As shown on the right, the 4:3 ratio produces most of the highest cross-validation *F1* scores. In this case, the *C* parameter range was narrowed to [0.03, 1.0], and the 3:1 and 2:1 ratios were removed from the undersampling options.



When results demonstrated poorer performance for a ratio, the ratio option was removed from the model optimization process, and the final model hyperparameters were tuned using one ratio.

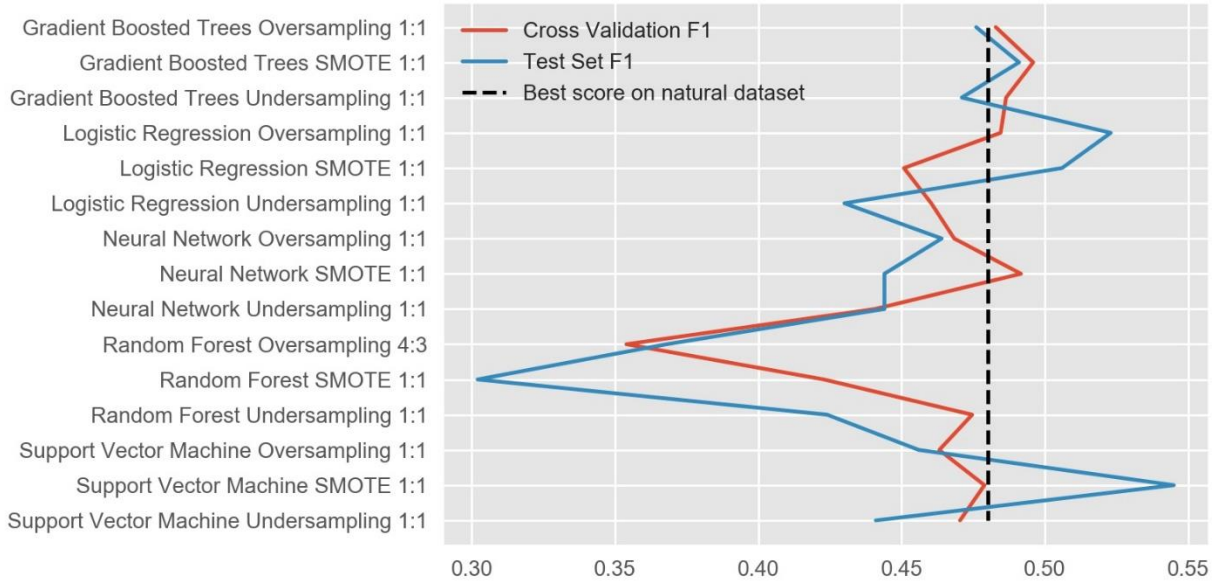
Tuning of the algorithm hyperparameters followed the same process I used with the natural dataset, by sequentially narrowing ranges and trimming options. A final hyperparameter and ratio configuration was selected for each model for evaluation on the test set.

Test set performance

After each model was tuned with cross-validation, each model was used to classify the held-out test set of 109 games. The plot below displays the test-set F1 score for each model. For comparison, the *F1* scores from cross-validation and test set from each model.

Notice that based on superior results during cross-validation and hyperparameter tuning, the 1:1 resampling ratio was used for nearly all of the models. Also note that test set scores were generally close to cross-validation scores, indicating good generalization performance.

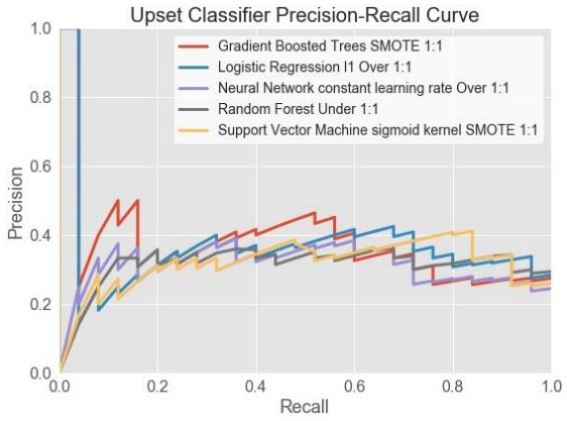
Recall the goal of training and testing with resampling methods, which was to produce a higher test set *F1* score than the 0.48 achieved when training models on the natural distribution. This score is indicated by the black dashed line. Several models surpass this benchmark.

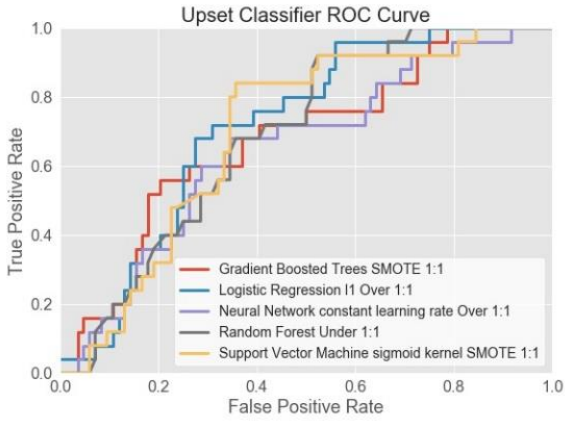


Below I display the top test set *F1* scores for each algorithm, along with additional classification scoring metrics. The top score for each algorithm using resampling bests the top score for each algorithm using the natural dataset.

Model	**F1**	**Recall**	**Precision**	**Accuracy**
Support Vector Machine SMOTE 1:1	0.545	0.84	0.404	0.679
Logistic Regression L1 oversampling 1:1	0.523	0.68	0.425	0.716
Gradient Boosted Trees SMOTE 1:1	0.491	0.56	0.438	0.734
Neural Network oversampling 1:1	0.464	0.64	0.364	0.661
Random Forest Undersampling 1:1	0.424	0.56	0.341	0.651

The plots below display precision-recall curves and receiver operating characteristic (ROC) curves for each model. Note that the 3 top models each exhibit superior performance at distinct points of the ROC curve. Gradient boosted trees, logistic regression, and support vector machine each maximize true positive rate (relative to the other algorithms) at distinct points in the ROC curve. This result suggests that combining the algorithms in an ensemble could be a way to maximize overall upset classification, but this idea was not tested further here.



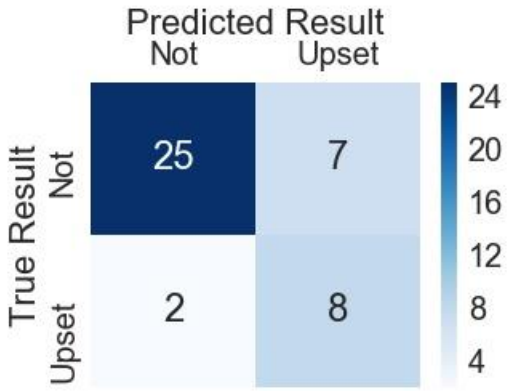


Future Matchup Classification

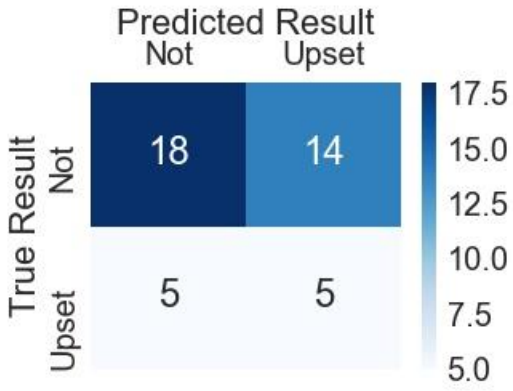
To illustrate a use case for the model predictions, the 42 upset-potential games from the 2017 tournament were classified. For this assessment, I used only the two top models from test set performance, logistic regression with oversampling and support vector machines using SMOTE. Classification scores and a confusion matrix for each are shown below. For comparison, the top 2017 *F1* score from my prior work was 0.60.

	F1	Recall	Precision	Accuracy
Logistic Regression	0.64	0.8	0.533	0.786
Support Vector Machine	0.345	0.5	0.263	0.548

Logistic Regression Confusion Matrix



Support Vector Machine Confusion Matrix



For the 2017 tournament, the Logistic Regression model with oversampling tops the prior benchmark of 0.60. The Logistic Regression also vastly outperforms the support vector machine. This outcome was surprising, since the support vector machine model had the top *F1* score when used to classify the test set. This result suggests the Logistic Regression model is the better choice for future use.

Comparison to 2017 Kaggle Contest Entries

To evaluate model performance against a competitive industry benchmark, I used team entries from the 2017 Kaggle March Machine Mania contest, which were posted after conclusion of the 2017 tournament. Although this contest has a different scoring criterion (log loss) and format (submission of probabilities for all possible tourney games), the entries for the contest provide a reasonable benchmark for data-driven prediction of NCAA tournament games.

Here I summarize the methodology used to conduct this comparison, with full details in a previous report available here:
https://github.com/mworles/capstone_one/blob/master/reports/capstone_one_final_2.0.pdf

For the Kaggle contest, each entry contains a probability estimate for each possible matchup in the tournament. My goal was to score each entry on classification of the same 42 "upset-potential" games from 2017 that I classified above. For each entry, I selected the 42 games, and converted the probabilities to upset predictions using a 0.50 cutoff. Each entry was scored with classification metrics (F1, recall, precision, accuracy), and I created a distribution of scores for each metric. This allowed me to compare my 2017 results to the distribution of scores from the 2017 Kaggle contest.

	F1	Recall	Precision	Accuracy
L1 Logistic Regression	0.64	0.8	0.533	0.786
Kaggle contest entries				
Percentile of my model	99.87	94.589	76.923	92.047
Kaggle contest mean	0.18	0.18	0.352	0.72
Kaggle contest max	0.667	1	1	0.857
Kaggle contest 25th percentile	0	0	0	0.738
Kaggle contest 75th percentile	0.308	0.2	0.5	0.762

For F1, my performance was at the 99th percentile and topped by only one Kaggle entry. For recall, my performance was at the 94th percentile. Closer inspection of the Kaggle scores revealed that 37 entries achieved recall above 0.80, but none of these entries had precision above 0.36. For accuracy, my performance was at the 92nd percentile.

Return-on-Investment (ROI) Analysis

To assess the "value" of this predictive model, I examined ROI from a simulated investment scenario for the 2017 NCAA Tournament. The conditions of the scenario were as follows.

1. Each gamble is a "money-line" bet on each upset-potential game.
2. Place a $100 bet on each game, using the model predictions to select the game winner.
3. Calculate the total of all returns and losses.
4. Calculate ROI as a percentage return as follows:

$$\text{ROI} = \text{Net return} / \text{Total wagered}$$

Under these conditions, the model predictions produced the following ROI:

$$\text{ROI} = 889 / 4200 = 21\%$$

This return is a slight improvement over the 17% ROI achieved with models trained on the natural dataset. Compared to those prior results, the best model from this project trained correctly classified one less true upset leading to slightly lower recall (0.8 vs 0.9), but correctly classified four more true non-upsets leading to higher precision (0.53 vs 0.45).

Client Recommendations

For bracket contest participants, the project provides one clear recommendation:

Pick Round 1 underdogs in games predicted as upsets by the model.
In 2017 these recommendations would have produced 23 correct predictions out of 24 games, for a 95.5% first-round accuracy rate in the 24 upset-potential games. The model predicted 3 of 4 true upsets, and correctly classified all 20 true non-upsets. Although this subset of games only constitutes 75% of games in the first round, correctly predicting all Round 1 upsets would provide bracket contest players with an early advantage in the leaderboard for most bracket pools, especially those that award bonus points for correctly predicting underdog winners.

For individual game wagers, the project provides a different recommendation:

Weight the investment budget towards Round 1 upset predictions. The model upset predictions were dramatically more accurate for games played in the first round of the 2017 tournament (95.5%) than the other rounds (56%). I would advise sports betting clients to weight investment budgets more heavily in the first round of the tournament.

Pad the investment returns with bets on high-confidence non-upsets. When inspecting the predictions for 2017, I found that no false positives came from examples where the upset probability was below 0.40 (19 correct / 19 games). These games also have some of the lowest return rates, but given this high accuracy, investors can mitigate their losses on the high-yield but more volatile upset predictions by allocating larger wagers towards these high-confidence non-upsets.

Summary

This project was a continuation of prior work that sought to develop accurate classification models for prediction of upsets in the NCAA tournament. This project addressed limitations of the prior project by testing a wider range of algorithms and using resampling to address the label imbalance. Each algorithm was tuned on an 80% training set using resampling and cross-validation. Algorithm configurations with the highest *F1* score in cross-validation were then compared on a held-out, 20% test set. Improvements in test set results indicated that resampling improved the performance of each classification algorithm. When used to classify games from a single tournament in 2017, an L1 Logistic Regression model achieved 0.64 *F1*, 0.80 recall, and 0.63 precision. This *F1* score was in the 1st percentile of F1 scores computed from Kaggle contest entries. Predictions from the model also generated 21% return in a simulated investment scenario, and correctly predicted 23/24 first-round tournament games between teams seeded at least 4 slots apart. Based on these results and the typical accuracy of bracket contest players in these games, this model could be used to offer improved upset guidance in future tournaments.

Ideas for Future Work

Several potential alternative strategies and questions about this problem create opportunities for future work.

Inspecting areas of poor performance/ensembling of algorithms. I noticed that on test set classification, the 3 top algorithms each appeared to have strengths in unique areas of the ROC curve space. In this project, I did not conduct in-depth analysis of examples that tended to be poorly classified by the various algorithms. It is possible that each algorithm tends to make unique mistakes, and that combining multiple algorithms into an ensemble could provide the best overall performance in classifying upsets. Future research could test this idea.

Using regression approaches to predict score margins. In this project I classify the final outcome of the game, which is a dichotomous event. When scores of a basketball game are close, the final outcome can swing unpredictably and depend on a few "chance" events that occur at the end of the game (e.g., an "incorrect" foul call, a "lucky" bounce on the rim). It is possible that I could achieve more accurate results by predicting the final scores of individual teams, or the final score margin between two teams. These predictions could be used to make bets "against the spread" as opposed to money-line bets.

Quantifying consistency and producing a data product. In theory, millions of annual tournament pool participants may be interested in accessing these predictions, especially if more extensive research could show that the model consistently produces accurate predictions and positive returns in single tournament years. A useful data product from this model would allow individual users to select potential tournament games and view the model-estimated prediction for the game winner with a probability value. Some internet outlets have produced similar products in recent years to estimate the progression probabilities for tournament teams (e.g., fivethirtyeight.com, Bing), but because these products largely mirror the "most likely" outcomes, few upsets are usually predicted. A data product focused specifically on identifying the tournament upsets could address this gap in the market.