T20 Cricket Predictions

Vibhu Pillai
Zach Buckley

Introduction

Cricket is a highly technical sport with a lot of factors that influence the outcome of the game. Scores differ based on venues. The pitch on which the game is played determines the output. For example a dry wicket would help the ball to come quicker to the batsman while it would assist the slower bowlers. The game is divided into 2 innings, one for each team. The team batting first sets a score with the stipulated 120 pitches. The team batting 2nd now has score at least one more than the runs scored by the 1st team. For our project we have tried to predict the outcome of the game based on a number of features available after the first inning is completed.

Cricket Background[1]

Cricket is played with two teams of 11 players each. Each team takes turns batting and playing the field, as in baseball. In cricket, the batter is a batsman and the pitcher is a bowler. The bowler tries to knock down the bail of the wicket. A batsman tries to prevent the bowler from hitting the wicket by hitting the ball. Two batsmen are on the pitch at the same time.

SCORING

The batters can run after the ball is hit. A run is scored each time they change places on the pitch. The team with the highest number of runs (typically in the hundreds) wins the match.

6 runs: A ball hit out of the field on a fly.
4 runs: A ball hit out of the field on a bounce.

OUTS (DISMISSALS)

Bowled out: Bowler knocks over (breaks) the wicket with a bowl.
Caught out: Fielder catches a batted ball on the fly
Run out: Fielder catches ground ball and throws it at the wicket, knocking it down before the batsman gets there.
Leg before wicket: Batsman's body interferes with a bowled ball that would hit the wicket.

THE FIELD

Circular, natural or artificial turf
Sizes vary from ground to ground. There are 11 players per team positioned around the oval.

THE GAME IS OVER WHEN

[1] "The basics of cricket, explained - Chicago Tribune." 15 Feb. 2015, https://www.chicagotribune.com/chi-cricket-basics-explanation-gfx-20150215-htmlstory.html. Accessed 19 Aug. 2019.

Sides take turns batting and fielding. Each at-bat, called an "over," comprises no more than six bowls per batsman. The fielding team must retire or dismiss 10 batsmen to end the innings (always plural). World Cup matches are limited to one inning per team and a limit of 50 overs per inning. Non-elimination games are limited to a single day. Elimination games are allowed a second day if needed.

The Dataset[2]

The dataset we found contained data of more than 6500 matches Each game with information such as the first innings score(also our key input), further inputs such as:

series_id
match_details
result
scores
date
venue
round
home
away
winner
win_by_runs
win_by_wickets
balls_remaining
innings1
innings1_runs
innings1_wickets
innings1_wickets
innings1_overs_batted
innings1_overs
innings2
innings2_runs
innings2_wickets
innings2_overs_batted
innings2_overs

Considering our goal is to predict the outcome of the game based on the information we have from the 1st innings so we ignore the information from the 2nd innings leaving us with :

Categorical: date, venue, round, home, away, winner, innings 1

[2] "T20 cricket matches | Kaggle." 12 Apr. 2017, https://www.kaggle.com/imrankhan17/t20matches. Accessed 19 Aug. 2019.

Continuous: win_by_runs, win_by_wickets, balls_remaining, innings1_runs, innings1_wickets, innings1_overs_batted

Note: First 3 columns are the original data that the dataset creator parsed to construct the remaining columns.

Preprocessing

Our initial data cleaning related to focusing the data we'd found on the defined problem. As we're looking to predict the outcome of the game based on information that's available at the end of the first inning, we can immediately drop all the features that are only available after the game is completed. This reduced the number of columns in the raw dataset by about 45%. We also dug through the dataset and identified missing data. For the most part, the only observations with missing data where those representing matches that had ultimately been abandoned. Removing the abandoned matches, further reduced the dataset by about 5% percent of the provided observations. This reduced collection of data was saved off as 'cleaned.csv' for later use.

Next, we set about preparing the data for training our models. Towards this end we wrote a function called load_cleaned, responsible for loading the data in clean.csv, setting the appropriate variable types, and performing some additional transformations on our features:

1. series_id: This features is effectively a foreign key connecting the data provided in models.csv to the data in series.csv. This could be an interesting way to pull in additional features for future iterations of this project. For our purposes though, the feature was dropped.
2. date: This feature could be expanded a number of ways to identify cyclic patterns in the target variables performance, but for our purposes we simply parsed the date as a datetime (pandas api), and encoded them as 'time since a date'. Expanding the date values into a number of categorical features is a tempting option recommended on stackoverflow[3], and may be an interesting way to improve the predictions in further iterations of the project.
    a. Hour of Day
    b. Day of Week
    c. Day of Month
    d. Day of Year
    e. Year
3. Home, away, innings1, innings2: these features are effectively repeating the same categorical variables, and I wanted to ensure they got encoded as being the same type. So, I combined the contents of the columns, and found all the unique values. In total there were about 340 teams mentioned. I then applied the same custom

[3] "Regression with Date variable using Scikit-learn - Stack Overflow." 9 May. 2013, https://stackoverflow.com/questions/16453644/regression-with-date-variable-using-scikit-learn. Accessed 19 Aug. 2019.

CategoricalDType to each of the columns, so that any given team "x" is encoded as 1 consistently within each of the columns.

4. Round: we discussed early on that round might be a good variable to drop, as it only exists for some of observed matches. This too would be an interesting expansion of the preprocessing currently being performed, though I'm uncertain of it's impact on the modeling results.
5. Target: we construct the target variable after reading in the cleaned dataset. The target variable is True, if the team that bowls first wins the game, and False otherwise. (Note: this variable is constructed in 'data.py' and saved in cleaned.csv)

Having setup all the variables types, and performed the transforms described above, we had a starting point for training our models, though admittedly some preprocessing would still be required before the models could do their thing. For the remaining preprocessing tasks, we will rely on sklearn's Pipeline construct, and several of the provided ColumnTransformers. Specifically, for our categorical variables we'll use a simple imputer that adds a 'missing' value for any observations missing that particular variable, followed by a OneHotEncoder.

For our numerical variables we'll apply a SimpleImputer transform that replaces missing values with that features median value. In experimenting with the dataset separately, we were able to determine that we aren't actually missing any entries for the numerical columns (after removing the abandoned matches), but for complete-ness it seemed reasonable to keep an imputer in the pipeline just in case. Our initial setup of the pipelines was based on a very useful portion of sci-kit learn's documentation[4]. From there we've pulled in a number of other sklearn tools, namely Cross Validated grid search, and randomized search in implementing the model training portions of our code. You'll notice in the code, that for the most part, each individual model's implementation includes the same copy-pasted preprocessing elements in the form of the sklearn Pipeline construct.

Models

Logistic Regression

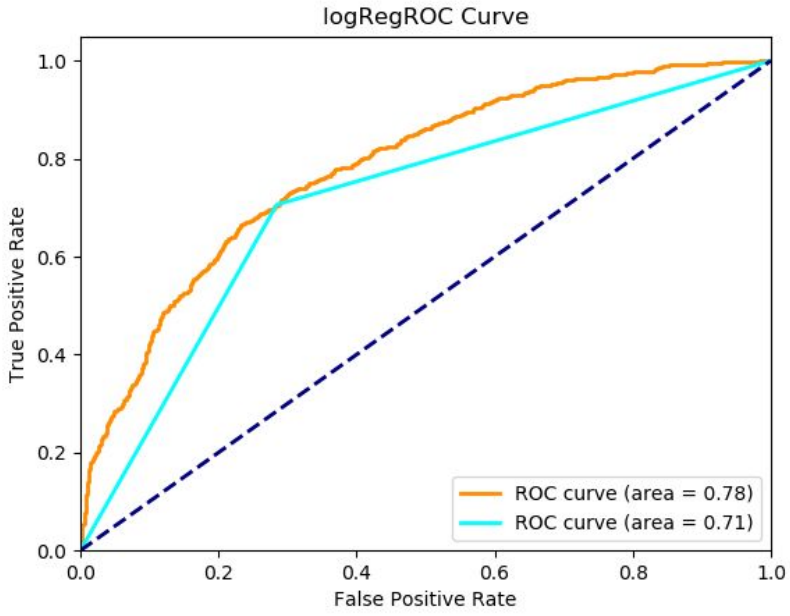
In training our logistic regression model, our initial thought was to attempt training an elastic-net logistic regression model, setting up the appropriate hyper-parameters for tuning to something in between Lasso and Ridge Regression that would be optimal for this problem. However, in attempting some of those initial experiments we had some issues getting the least squares algorithm to converge on a solution. This appears to be caused by a degrees of freedom problem between the number of features introduced by the one-hot encoding, and the number

[4] "Column Transformer with Mixed Types — scikit-learn 0.21.3 ...." http://scikit-learn.org/stable/auto_examples/compose/plot_column_transformer_mixed_types.html. Accessed 19 Aug. 2019.

of observations available in the training set (especially when further reduced using 5-fold cross-validation). As such we proceeding using a Lasso (or L1 Regularization) Regression, since that optimizes the training algorithm for removing features as quickly as possible. To further combat the convergence issues, we set the sklearn solve algorithm to use double the default number of iterations, and increased the algorithms default tolerance by a factor or 10.

While working through various iterations of training our regression based model, the sklearn documentation page on it's LogisticRegressionCV[5] class was an extremely valuable resource. The Logistic Regression model, even with double the default number of allowed iterations was pleasantly quick to train, making it fairly easy to proceed using trial and error as we worked through the issue with convergence.

LogReg Confusion Matrix (vs. test data)		Actual Class	
		True	False
Predicted Class	True	677	269
	False	281	612



[5] "3.2.4.1.5. sklearn.linear_model.LogisticRegressionCV — scikit-learn ...." http://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html. Accessed 19 Aug. 2019.

Random Forest

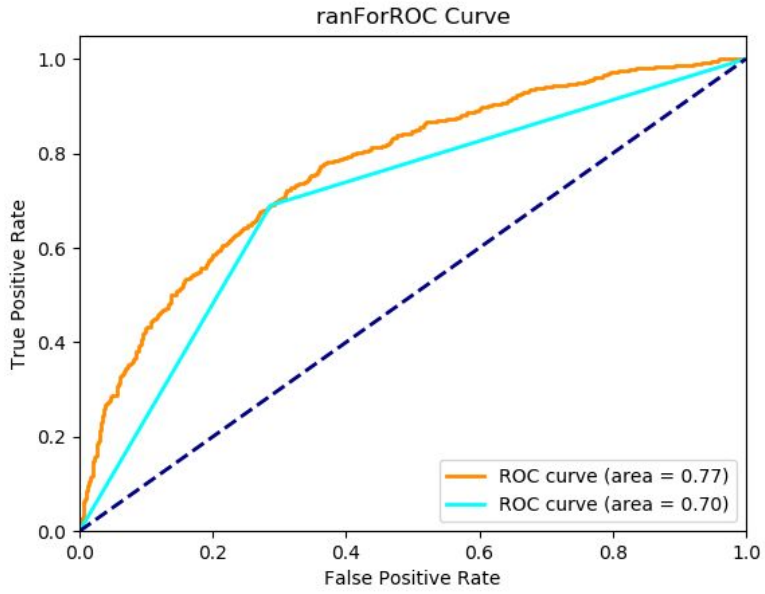
Training our Random Forest Classification model took very little time to get into, as we could copy most of the preprocessing code from the preprocessing we'd setup for the Logistic Regression model. Random forest classification models of course have an intimidating array of hyperparameters that can be tuned, we decided after some initial experimentation to do utilize another sklearn utility for performing a stratified 5-fold cross validated grid search to identify the correct parameters[6]. We limited the parameters of interest to the number of trees: 10, 100, 200, and 500 trees, and the maximum depth of those trees: 25, 50, 100. Sklearn proved to be a powerful tool for this, as with very little actual code, we attempted 12 different hyperparameter combinations, 5 times each. (training a total of 60 different random forest models). The most optimum model got a Cohen's Kappa score of 0.403, using 200 trees with max_depth of 50.

Results are:

RF Confusion Matrix (vs. test data)		Actual Class	
		True	False
Predicted Class	True	675	271
	False	269	598

An interesting expansion of our experiments with a Random Forest based model, would be to look into the use of different Boosting or Bagging techniques in training the trees. As is, the model we have is using sklearn's RandomForestClassifier's[7] default settings, which best I can tell will not use any Boosting or Bagging techniques (it looks like those involve other model classes provided through the sklearn api).

[6] "sklearn.model_selection.GridSearchCV — scikit-learn 0.21.3 ...." http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. Accessed 19 Aug. 2019.
[7] "3.2.4.3.1. sklearn.ensemble.RandomForestClassifier — scikit-learn ...." http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. Accessed 19 Aug. 2019.



SVM Classifier

As with Random Forest, we were able to start experiments with sklearn's SVC[8] (Support Vector Machine Classifier) implementation very rapidly, due to a common chunk of preprocessing Pipeline code being used. The classification problem is likely not linearly separable, but regardless with ~1,860 features, after the One-Hot Encoding expansion, it would be impossible to visualize which provided kernel function would give us the optimal output. So instead, we've applied our sklearn GridSearchCV functionality again, this time attempting to seek out the best kernel function for us to use.

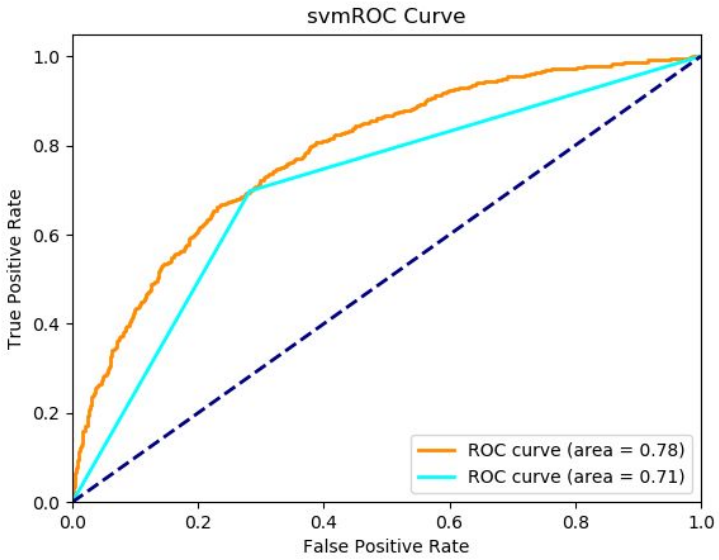
Our resulting final model uses the Radial Basis Function Kernel, provided by sklearn's api. According to sklearn documentation, the kernel function[9] is:

$$k(x_i,\ x_j) = exp\left[-\tfrac{1}{2} dist\left(\frac{x_i}{lengthScale},\ \frac{x_j}{lengthScale}\right)\right]$$

Not that we really gleaned any insights from that, but it was interesting that the functions were at least available for further inspection or analysis if needed. This optimal model did reasonably well on the test set scoring a Cohen's Kappa of 0.415, and an Accuracy of 0.708.

[8] "sklearn.svm.SVC — scikit-learn 0.21.3 documentation." http://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html. Accessed 19 Aug. 2019.
[9] "sklearn.gaussian_process.kernels.RBF — scikit-learn 0.21.3 ...." http://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.RBF.html. Accessed 19 Aug. 2019.

SVM Confusion Matrix (vs. test data)		Actual Class	
		True	False
Predicted Class	True	679	267
	False	262	605



Naive Bayes Classification[10]

Training Naive Bayes Classifier using the sklearn library proved to be more challenging than I expected. I had anticipated a NaiveBayes Classifier existing with sklearn, but instead the sklearn library provides a couple Naive Bayes based models, tailored to data meeting certain criteria:

- GaussianNaiveBayes[11]: intended for continuous features which align with gaussian distributions

[10] "1.9. Naive Bayes — scikit-learn 0.21.3 documentation." http://scikit-learn.org/stable/modules/naive_bayes.html. Accessed 19 Aug. 2019.
[11] "sklearn.naive_bayes.GaussianNB — scikit-learn 0.21.3 documentation." http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html. Accessed 19 Aug. 2019.

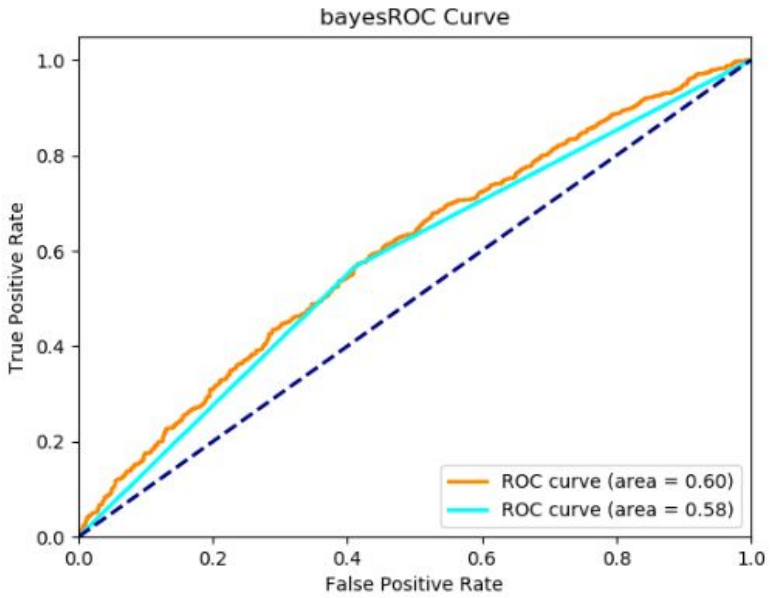
- MultinomialNaiveBayes[12]: intended for discrete, features like 'number of times x occurs'
- BernoulliNaiveBayes[13]: intended for binary categorical features, with align with Bernoulli distributions

My assumption is that to truly build a purely Naive Bayes classifier in sklearn, you would need to somehow combine these models using Ensemble modeling techniques. For our purposes, I initially decided to attempt to train a GaussianNaiveBayes model. However, I ran into trouble when trying to transform the data inputs. GaussianNaiveBayes apparently requires the input to be a numpy array, rather than a pandas model, and causing that transformation to occur in the context of an sklearn pipeline proved quite challenging. I discovered after some additional experimenting, that I could drop the non-categorical features. And provide the resulting pandas based dataset to the BernoulliNaiveBayes model. This model would obviously suffer from not having influence from the removed features, but after several hours… a better solution wasn't close to being available, proving not to be an option for this attempt at the problem. A further iteration of this experiment could certainly revisit our attempt to train this model, and no doubt improve it quite drastically.

The BernoulliNaiveBayes model doesn't really have any hyper-parameters that have to be tuned, so this implementation ended up requiring a slight modification to our existing preprocessing pipeline code, and adding a single line to initialize the BernoulliNaiveBayes model. Anything that easy… of course doesn't yield stunning results though. This model is the worst performing of the bunch, though certainly due to having less features available for it's training/evaluation of the problem. The final model got an accuracy of 0.578, and a Cohen's Kappa score of 0.155. This prompted me to verify just what our null accuracy rate is for the dataset, and sure enough it's 0.523. So… the model only has about 5% better accuracy than us simply guessing.

Bayes Confusion Matrix (vs. test data)		Actual Class	
		True	False
Predicted Class	True	556	390
	False	375	492

[12] "sklearn.naive_bayes.MultinomialNB — scikit-learn 0.21.3 documentation." http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html. Accessed 19 Aug. 2019.
[13] "sklearn.naive_bayes.BernoulliNB — scikit-learn 0.21.3 documentation." http://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.BernoulliNB.html. Accessed 19 Aug. 2019.


bayesROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 0.60)
ROC curve (area = 0.58)

K Nearest Neighbors

K Nearest Neighbors wasn't initially on the list of models I intended to try and train for this problem set, though not for any particularly good reason… just hadn't considered it. Setting up the training was fairly straight forward. The obvious hyperparameter to train for this model, is the K, or the number of neighbors to evaluate before making a classification. I initially attempting to tune this parameter with very low numbers, handing GridSearchCV[14] lists like [1,3,5,7], and the like. After utilizing RandomizedSearchCV[15], in my experiments with the Neural Network based models we'll get to in a bit, I came back and revisited this model's training. I hadn't really considered before why a gradient descent style search algorithms aren't readily available for tuning hyperparameters in sklearn. After some experimenting, and some further digging, I stumbled across a stack overflow[16] post describing why sklearn wouldn't provide that. In short… hyper-parameter optimization isn't a smooth function… which totally makes sense. And made me rethink my earlier approach of narrowing down the amount of neighbors to use for k-means through what amounted to a naive gradient descent, where I would increasingly narrow the options for the GridSearchCV class to try.

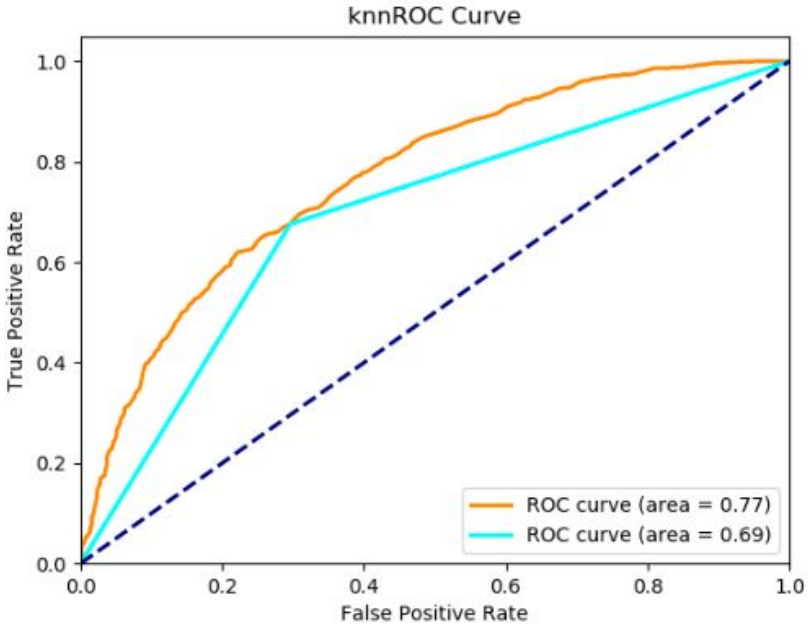
Anyway… applying the randomized search quickly got me out of my "surely it can't more than 20" rutt, by generating a KNN model using K=317, which performed, if not as well, at least comparably to the other models we've been training, getting an accuracy of 0.691, and Cohen's Kappa of 0.380.

KNN Confusion Matrix (vs. test data)		Actual Class	
		True	False
Predicted Class	True	666	280
	False	281	586

[14] "sklearn.model_selection.GridSearchCV — scikit-learn 0.21.3 ...." http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html. Accessed 19 Aug. 2019.

[15] "sklearn.model_selection.RandomizedSearchCV — scikit-learn 0.21.3 ...." http://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html. Accessed 19 Aug. 2019.

[16] "sklearn: Hyperparameter tuning by gradient descent? - Stack Overflow." 15 Apr. 2017, https://stackoverflow.com/questions/43420493/sklearn-hyperparameter-tuning-by-gradient-descent. Accessed 19 Aug. 2019.


knnROC Curve
True Positive Rate
False Positive Rate
0.0
0.2
0.4
0.6
0.8
1.0
ROC curve (area = 0.77)
ROC curve (area = 0.69)

Multi-Layer Perceptron Classification

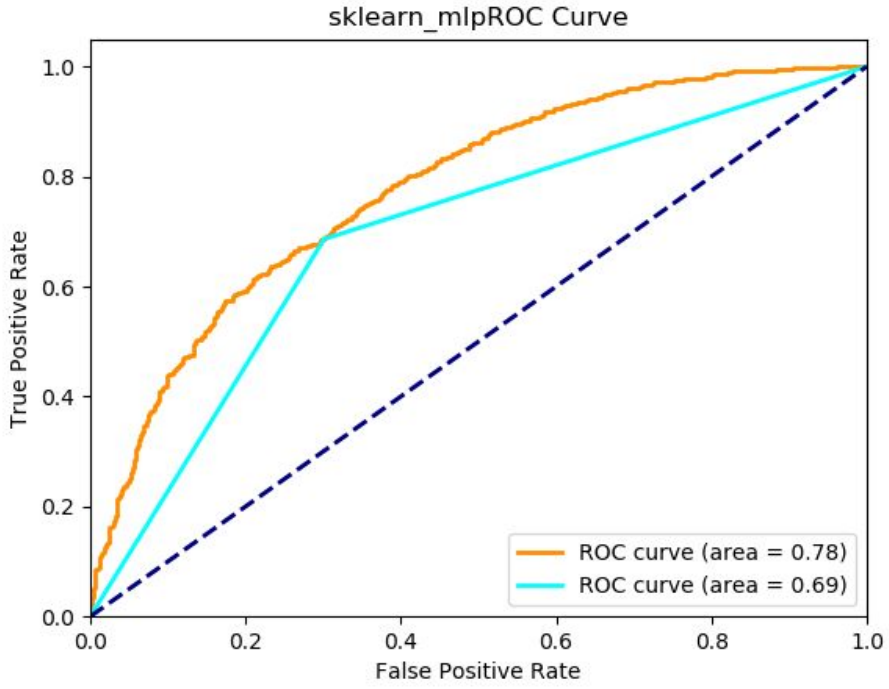
There is code for 2 different MLP models in our project repository. That is because, with the ease of switching between (most) of sklearn's Estimators[17], I was curious to attempt using their MLPClassifier type, prior to looking at the (probably) much more robust Neural Network options available in the Keras Framework.

Sklearn based Multi-layer Perceptron

In short, it took a very long time to train the network, and tuning hyper-parameters, like activation function, solver algorithm, learning rate, number of hidden layers, number of neurons (per hidden layer), and number of iterations.. Proved to be quite painful. So, I put about an afternoon's worth of time into messing around with different parameters for the sklearn based MLP model, and finally decided to just move on to the keras model, since it was taking so much time to try tuning the MLP model further. The model I did end up with from my experiments with skearn's implementation, performs moderately relative to the rest, giving us an accuracy of 0.693, and Cohen's Kappa of 0.385. Considering the amount of time it took me to tune this model's parameters to get it to a point where it performed that well, it isn't something that would normally be worth doing.

MLP Confusion Matrix (vs. test data)		Actual Class	
		True	False
Predicted Class	True	661	285
	False	272	595

[17] "sklearn.base.BaseEstimator — scikit-learn 0.21.3 documentation." http://scikit-learn.org/stable/modules/generated/sklearn.base.BaseEstimator.html. Accessed 19 Aug. 2019.



Keras based Multi-layer Perceptron

Setting up Keras to run in CPU land is a fairly trivial thing, but to properly experiment with the library I wanted to ensure that Keras's backend (Tensorflow) was setup with access to my laptop's GPU, and the required CUDA libraries to really make use of it. I was able to find some fairly useful help in setting it all up from the Keras and Tensorflow websites, in particular I found a list of required libraries[18], which then made it much more manageable.
After working through getting my GPU accelerated Tensorflow backend working, I began figuring out what portions of my preprocessing code (heavily using sklearn) was compatible. Luckily, keras and sklearn are designed to be interoperable, at least to an extent. I was able to use classes Keras provides in it's api, to better connect the two libraries, specifically KerasClassifier[19]. By wrapping a Keras model in a KerasClassifier, you can feed inputs to the Keras model after having preprocessed the data using sklearn's Pipeline's. Excellent. I just saved myself reworking a fairly significant part of the preprocessing code.

I setup the MLP network in Keras, so that by default it would have 2 hidden layers, with 455 neurons each, and use the 'relu' activation, or transfer function. With a dropout rate of 0.0 (ie. no dropout by default). I also used the rmsprop optimizer, categorical crossentropy loss function, and accuracy metrics. Most of these settings were inspired by the documentation Keras

[18] "GPU support | TensorFlow." https://www.tensorflow.org/install/gpu. Accessed 19 Aug. 2019.
[19] "Scikit-learn API - Keras Documentation." http://keras.io/scikit-learn-api/. Accessed 19 Aug. 2019.

provides, including examples, on how to build simple sequential models[20] [21]. Our model construction function is setup so that the sklearn GridSearchCV, and RandomSearchCV classes can pass in 'hyper-parameters' when searching for the optimal, or at least better settings. The hyper-parameters I actually ended up searching through, were:

- Epochs
- Dropout_rate
- Extra_layers
- Num_neurons

As we were using the RandomizedSearchCV class from sklearn to do our searching, I provided not a specific list of inputs for the search algorithm to try, but instead a definition of the distribution to pull possible numbers from. To do this I used two functions from scipy, namely randint[22], and uniform[23] as needed for continuous and discrete hyperparameter values. We'll use 5 fold cross validation (stratified splitting) through the RandomizedSearchCV api, and sample 50 different hyper-parameter combinations, keeping the best one as our final model. As we're using 5-fold cross validation, this means we'll train about 250 MLP models, while attempting to tune our hyperparameters.

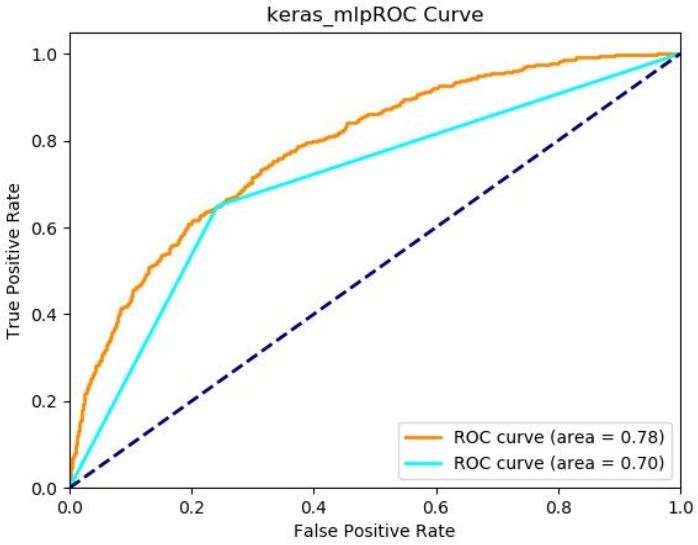
I ran into one hiccup… after several models were trained through the RandomizedSearchCV algorithm, the underlying keras/tensorflow/cuda (not sure which) library would fail, due to an Out of Memory Error, referring to GPU memory being unavailable for the next training task. It took me a while to work through that one, but eventually I figured out through a stackoverflow posting that you can force the tensorflow api to reinitialize itself each time the RandomizedSearchCV resets the model[24]. This likely slows the process down a bit, but at least allows it to continue through to completion.

At long last, I had a Keras based MLP model successfully trained, and with tuned hyperparameters, based on 5-fold cross validated randomized searching. The model performed relatively well, though not (as I'd hoped) drastically better than the other models we'd trained, ending up with an accuracy of 0.706, and cohen's kappa of 0.409.

---

[20] "Getting started with the Keras Sequential ...." http://keras.io/getting-started/sequential-model-guide/. Accessed 19 Aug. 2019.
[21] "Mnist mlp - Keras Documentation." https://keras.io/examples/mnist_mlp/. Accessed 19 Aug. 2019.
[22] "scipy.stats.randint — SciPy v0.15.1 Reference Guide." https://docs.scipy.org/doc/scipy-0.15.1/reference/generated/scipy.stats.randint.html. Accessed 19 Aug. 2019.
[23] "scipy.stats.uniform — SciPy v1.3.0 Reference Guide - SciPy.org." 17 May. 2019, https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.uniform.html. Accessed 19 Aug. 2019.
[24] "What do I need K.clear_session() and del model for (Keras with ...." 26 Feb. 2019, https://stackoverflow.com/questions/50895110/what-do-i-need-k-clear-session-and-del-model-for-keras-with-tensorflow-gpu. Accessed 19 Aug. 2019.

MLP Confusion Matrix (vs. test data)		Actual Class	
		True	False
Predicted Class	True	684	262
	False	273	594



Model Comparison and Conclusion

To compare our models, we're using both accuracy and Cohen's Kappa. We chose to use Cohen's Kappa in addition because it takes null accuracy into account. In fact all of the Cross-Validated parameter tuning searches performed in training our models, were done using a cohen's kappa scoring function, instead of the default accuracy function, to ensure that the null accuracy rate was accounted for in the optimizations. We know that test data has a null accuracy rate of ~52%. From there we can use the following table, which summarizes the model performance using the accuracy and Cohen's Kappa metrics:

Model	**Accuracy**	**Cohen's Kappa**
Logistic Regression	0.711	0.421
Keras based MLP	0.706	0.409
Support Vector Machine	0.708	0.415
Random Forest	0.702	0.403
Sklearn based MLP	0.693	0.385
KNearestNeighbors	0.691	0.380
Bernoulli Naive Bayes	0.578	0.155

It's a little disappointing to discover at the end of all this that Logistic Regression, the first model we implemented, as it seemed easier to get up and running, has the best performance on the test set. Our Naive Bayes model is bringing up the rear of the pack, but we have to remember that, unlike the other models, it is working with only the categorical data.

Bonus Ensemble Models (Stacking)

Having completed much of the work above, we were inspired to attempt using another 'layer' of models on top of these model predictions to see if combining the model outputs could yield better results on the test set. We started by building up an intermediate dataset, by performing predictions from each of the previously trained models, and saving the results from each model as new features in a pandas dataframe. We did this transformation to our intermediate dataset for both the training and testing features. Then used the intermediate data (for inputs) along with

the existing target vectors, as training and test data for a whole new list of models. The results we collected in terms of accuracy, and cohen's kappa for each model are below:

Model	**Accuracy**	**Cohen's Kappa**
Stacked Bernoulli Naive Bayes	0.707	0.414
Stacked KNearestNeighbors	0.707	0.413
Stacked Random Forest	0.707	0.412
Stacked Logistic Regression	0.707	0.412
Stacked Support Vector Machine	0.707	0.412