Supervised ML - Classification Assignment:

Experiment: Can we classify NBA Player salaries from John Hollinger's advanced NBA Statistics?

Objective

The purpose of this report, and overall study, is to experiment, checking to see if it is possible to classify the salary of a NBA Player based on a relationship between player salary and player performance using John Hollinger's NBA individual advanced statistics (2002-03 to 2017-18 seasons) and a list of player salaries (2003 – 2019 seasons). Based on raw numbers, player salaries will be grouped into 4 different classes, basically representing quartiles and we will see if advanced statistics do a good job of predicting the salary quartile of players.

A possible business implementation of this model, if successful in being able to classify salaries by performance, is to aid in value based contracting of players – essentially a version of 'moneyball'.

The Data

The data used for the analysis is taken from 2 separate Kaggle datasets, that are processed (using Pandas) and then merged. Each is a singe csv file

- The first summarizes John Hollinger's advanced NBA statistics from the 2002 – 03 season to the 2017 – 18 season. The data set is already cleaned, containing no zero, N/A etc data and contains 5404 row entries. Column entries include:
  - Rank: that player's PER rank for that given season
  - ts%: True Shooting Percentage - what a player's shooting percentage would be if we accounted for free throws and 3-pointers. True Shooting Percentage = Total points / [(FGA + (0.44 x FTA)]
  - ast: Assist Ratio - the percentage of a player's possessions that ends in an assist. Assist Ratio = (Assists x 100) divided by [(FGA + (FTA x 0.44) + Assists + Turnovers]
  - to: Turnover Ratio - the percentage of a player's possessions that end in a turnover. Turnover Ratio = (Turnover x 100) divided by [(FGA + (FTA x 0.44) + Assists + Turnovers]
  - usg Usage Rate - the number of possessions a player uses per 40 minutes. Usage Rate = {[FGA + (FT Att. x 0.44) + (Ast x 0.33) + TO] x 40 x League Pace} divided by (Minutes x Team Pace)
  - orr: Offensive rebound rate
  - drr: Defensive rebound rate
  - rebr: Rebound Rate - the percentage of missed shots that a player rebounds. Rebound Rate = (100 x (Rebounds x Team Minutes)) divided by [Player Minutes x (Team Rebounds + Opponent Rebounds)]
  - per: Player Efficiency Rating is the overall rating of a player's per-minute statistical production. The league average is 15.00 every season.
  - va: Value Added - the estimated number of points a player adds to a teamâ€™s season total above what a 'replacement player' (for instance, the 12th man on the roster) would produce. Value Added = ([Minutes * (PER - PRL)] / 67). PRL (Position Replacement Level) = 11.5 for power forwards, 11.0 for point guards, 10.6 for centers, 10.5 for shooting guards and small forwards

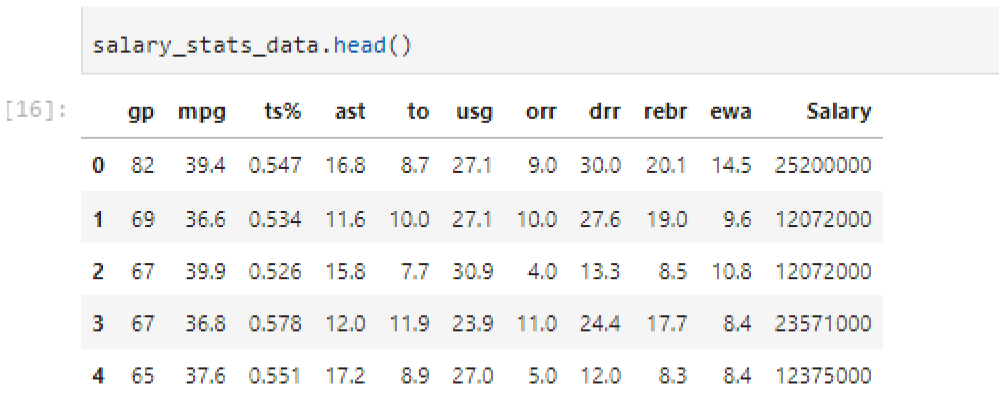
    - ewa: Estimated Wins Added - Value Added divided by 30, giving the estimated number of wins a player adds to a team's season total above what a 'replacement player' would produce.
- The second is a record of player salaries, from the 2003 – 2019 seasons inclusive. Again, the data was pre-processed (to an extent) and contained the following features:
    - Player name
    - Team
    - Season
    - Salary

Outlier data was not originally removed from either dataset as we did not want to remove potentially relevant data before looking closer.

Given that the features per, va, and ewa are quite similar in that they describe a players contribution, the va and per features were dropped for the analysis to avoid any effects of multicollinearity. Estimated Wins Added (EWA) was used as the sole summary statistic in the analysis – to note, it could have been either of the 3 (PER, VA, EWA), EWA was chosen for its simplicity and built in scaling (EWA is a scaled down version of VA, which is calculated from PER)

To make a final dataset, ready for exploratory analysis, we then combined the 2 separate datasets (player advanced statistics and salary data) into a single data frame. To complete this, some intermediate steps were required:

- Had to reformat the season column in the advanced stats data (ie: to read 2002 instead of 2002-03)
- Had to drop the 2002 season from the advanced stats data, as it does not exist in the salary data set
- Had to drop the 2018 and 2019 seasons from the salary data set, as they did not exist in the advanced stats data set
- Merge the data frames
- Drop the player and season columns: this left us with a data frame that contained only advanced stats (independent variables) and salary (the predictor)



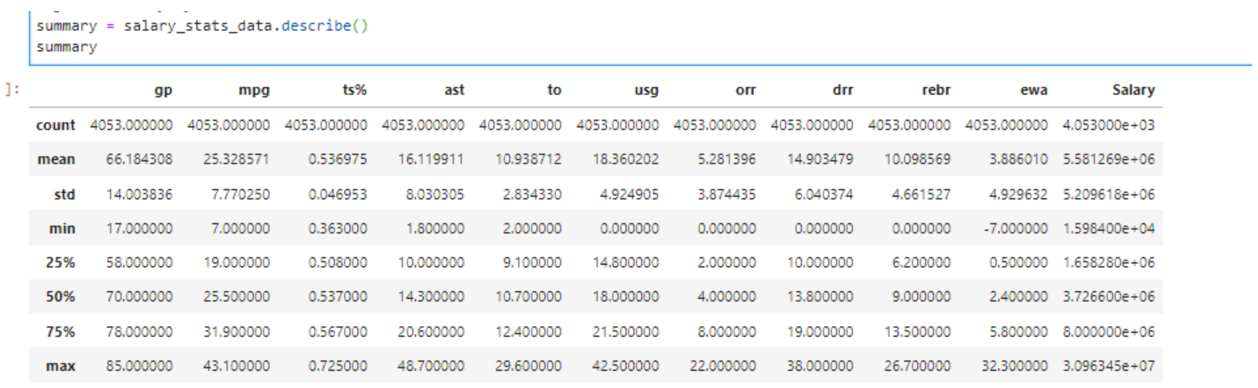
```
salary_stats_data.head()
```

[16]:

	gp	mpg	ts%	ast	to	usg	orr	drr	rebr	ewa	Salary
0	82	39.4	0.547	16.8	8.7	27.1	9.0	30.0	20.1	14.5	25200000
1	69	36.6	0.534	11.6	10.0	27.1	10.0	27.6	19.0	9.6	12072000
2	67	39.9	0.526	15.8	7.7	30.9	4.0	13.3	8.5	10.8	12072000
3	67	36.8	0.578	12.0	11.9	23.9	11.0	24.4	17.7	8.4	23571000
4	65	37.6	0.551	17.2	8.9	27.0	5.0	12.0	8.3	8.4	12375000

<u>Exploratory Analysis</u>

The first thing done was to get a description of the data. Since all the data was numeric, it is nice to look at the mean, median and different quartiles of the data, to see how it is spread out.



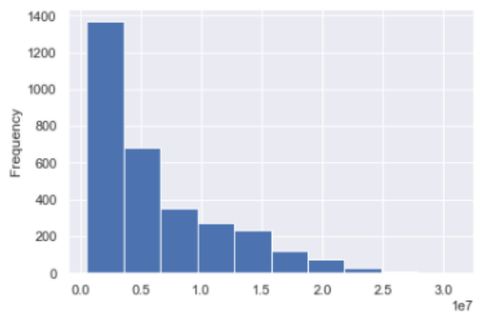
```
summary = salary_stats_data.describe()
summary
```

	gp	mpg	ts%	ast	to	usg	orr	drr	rebr	ewa	Salary
count	4053.000000	4053.000000	4053.000000	4053.000000	4053.000000	4053.000000	4053.000000	4053.000000	4053.000000	4053.000000	4.053000e+03
mean	66.184308	25.328571	0.536975	16.119911	10.938712	18.360202	5.281396	14.903479	10.098569	3.886010	5.581269e+06
std	14.003836	7.770250	0.046953	8.030305	2.834330	4.924905	3.874435	6.040374	4.661527	4.929632	5.209618e+06
min	17.000000	7.000000	0.363000	1.800000	2.000000	0.000000	0.000000	0.000000	0.000000	-7.000000	1.598400e+04
25%	58.000000	19.000000	0.508000	10.000000	9.100000	14.800000	2.000000	10.000000	6.200000	0.500000	1.658280e+06
50%	70.000000	25.500000	0.537000	14.300000	10.700000	18.000000	4.000000	13.800000	9.000000	2.400000	3.726600e+06
75%	78.000000	31.900000	0.567000	20.600000	12.400000	21.500000	8.000000	19.000000	13.500000	5.800000	8.000000e+06
max	85.000000	43.100000	0.725000	48.700000	29.600000	42.500000	22.000000	38.000000	26.700000	32.300000	3.096345e+07

Next, outliers were removed from both the games p[played (gp) and salary data. Note that, for the sake of analysis only the low outliers were removed from each feature

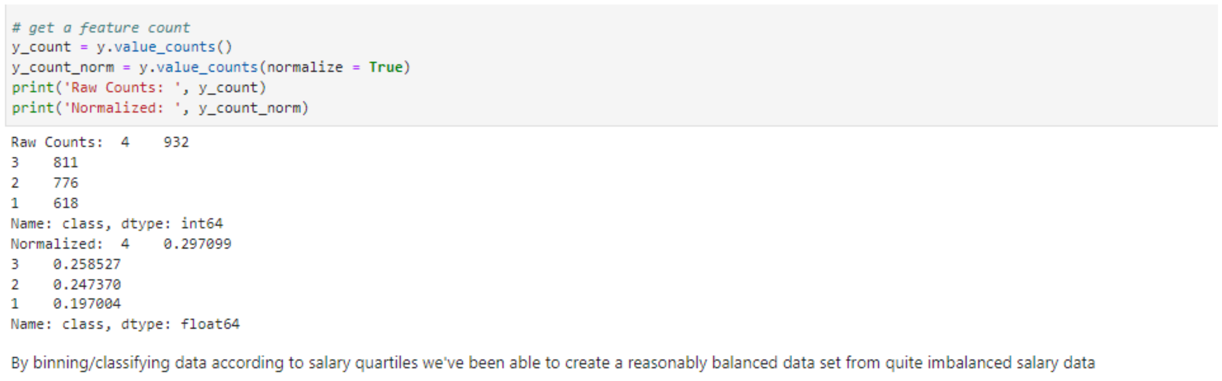
- Since value added is proportional to being in the game, we removed and values below the 25% value for games played – 1.5*IQR. This way, the analysis wouldn't be skewed too much by players who are being paid but adding little value due to long term injury
- Salary Data: to eliminate data from players of short term (10 day, 1 month) contracts (usually picked up due to injury problems with long term contracted players) we removed any rows where the salary data was below the 2003 players minimum salary ($564,000)
- Finally, for all other features, to ensure there were no erroneous data, rows were eliminated for all other features in the value was less than or equal to zero (since all other features were rates, it is impossible for a rate to truly be zero if playing – if they are, it is likely due to being rounded)

Next, the salary data was binned by quartile. Salary data was split into 4 different classes based on which quartile the salary fell into. As can be seen below, this did a great job of taking a dataset that is quite lognormally distributed (most salaries are clustered towards the lower end) and balancing out the data set:



As we can see, a disporoprtionately large amount of the salry data is skewed towards lower salaries (there are a lot more role players than stars)

By ‘binning’ the data, we were able to much better balance the data set. The data set was labeled 1, 2, 3, 4, based on the quartile that the salary data fell into (4 being the highest, 1 being the lowest)

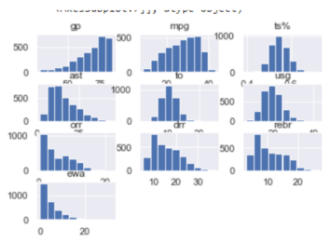


```
# get a feature count
y_count = y.value_counts()
y_count_norm = y.value_counts(normalize = True)
print('Raw Counts: ', y_count)
print('Normalized: ', y_count_norm)
```

```
Raw Counts:  4    932
3    811
2    776
1    618
Name: class, dtype: int64
Normalized:  4    0.297099
3    0.258527
2    0.247370
1    0.197004
Name: class, dtype: float64
```

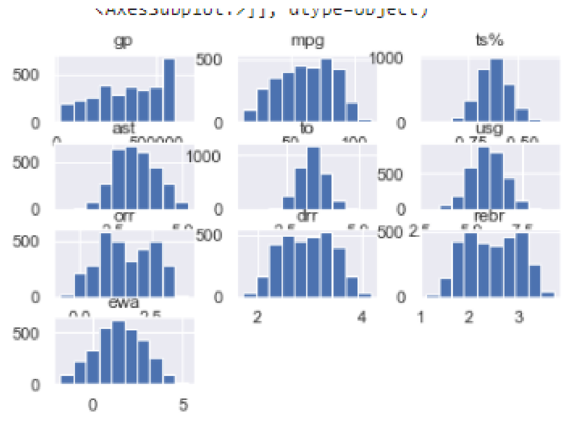
By binning/classifying data according to salary quartiles we've been able to create a reasonably balanced data set from quite imbalanced salary data

Next, we looked at the distribution of each of the features:



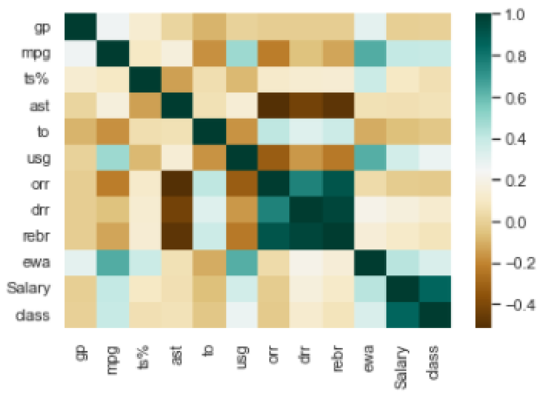
In my literature reviews (Python Machine Learning) it suggests that standardizing data is more beneficial than min/max scaling for classification mehtods. So, we will transform data to get the distributions normal and then use standard scaling

Note that each feature is distributed differently. In my literature reviews, it is mentioned that for classification algorithms Standard Scaling works better than Min-Max scaling. I then used a BoxCox transformation to get each feature as close as possible to normally distributed. After Bocxox transformation:



Now, most features are a little better – much closer to being normally distributed. Certainly close enough for Standard Scaling to apply

Finally, a correlation heatmap was built, to show the correlations between features and the correlations between the features and the target variable (salary):



Salary has somewhere between low and high correlation with each of the variables. this will liekly increase when binning

The data was then standardized – transformed using the sklearn StandardScaler and then split into a train/test split of 70/30 using the stratified shuffle split to maintain balance in each data set. As seen below, each data set kept its balance

```
x train:  (2195, 10)
x test: (942, 10)
y train:  (2195,)
y test:  (942,)
```

```
[31]: # check value counts of y_train and y_test, normalized
      print('y train: ', y_train.value_counts(normalize = True))
      print('y test: ', y_test.value_counts(normalize = True))
```

```
y train:  4    0.297039
3    0.258770
2    0.247380
1    0.196811
Name: class, dtype: float64
y test:  4    0.297240
3    0.257962
2    0.247346
1    0.197452
Name: class, dtype: float64
```

As expected, the data sets kept their class distribution through the split

Data Analysis

Logistic Regression

The first classification model that was fit on the transformed data was a Logistic Regression, using the LogisticRegressionCV model. Hyperparameters were set up as follows:

Cs = 10

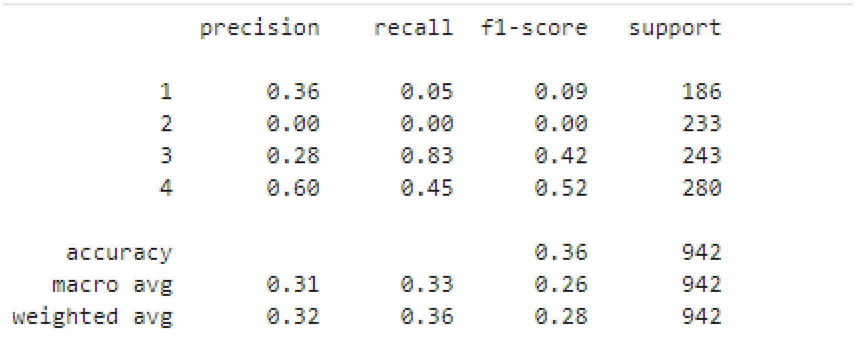
Cv = 4

Penalty = l2

Solver = liblinear

Overall, the logistics regression did a relatively poor job of modeling the data – see the following classification report. Notably:

- Recall (the number of actual positives predicted correctly) is terribly low, especially for lower salary classes (1, 2)
- Accuracy metrics improve in higher salary classes, but still are not great
- Reasons will be covered in the final discussion

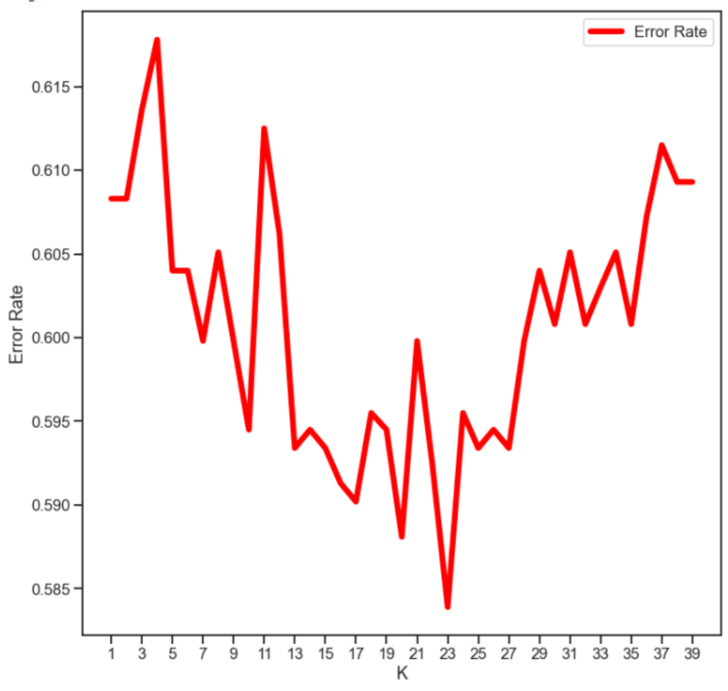


```
              precision    recall  f1-score   support

           1       0.36      0.05      0.09       186
           2       0.00      0.00      0.00       233
           3       0.28      0.83      0.42       243
           4       0.60      0.45      0.52       280

    accuracy                           0.36       942
   macro avg       0.31      0.33      0.26       942
weighted avg       0.32      0.36      0.28       942
```

Logistic Regression does not appear to suitably model the data. We will attempt 3 other algorithms to see if distance or tree-based algorithms do a better job of modeling/predicting than a loglinear model

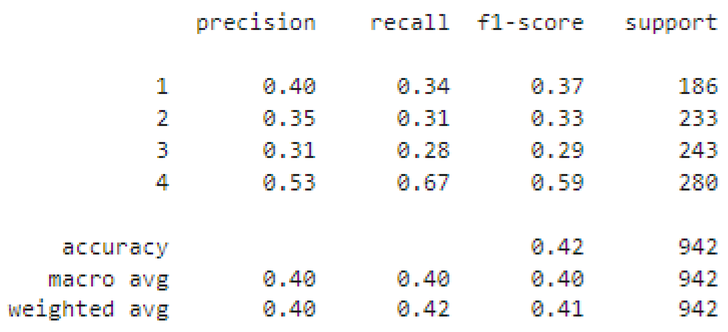
K-Nearest Neighbors

Next, we ran a KNN algorithm on the data, simulating for k's in the range of 1 to 40. Based on the error rate, we will determine the best value of k, which based on the model appears to be k = 23:



Again, running the algorithm, to determine accuracy, for the best model (k = 23), we find that the overall accuracy of the model increases -> it appears that distance-based models work a little better on this dataset. See the classification report below:

- Both precision/recall and accuracy are much higher at lower salary classes
- Overall accuracy is a mixed-bag at higher levels (a little better at class 4, but worse at class 3)
- Reasons will be covered in the final discussion

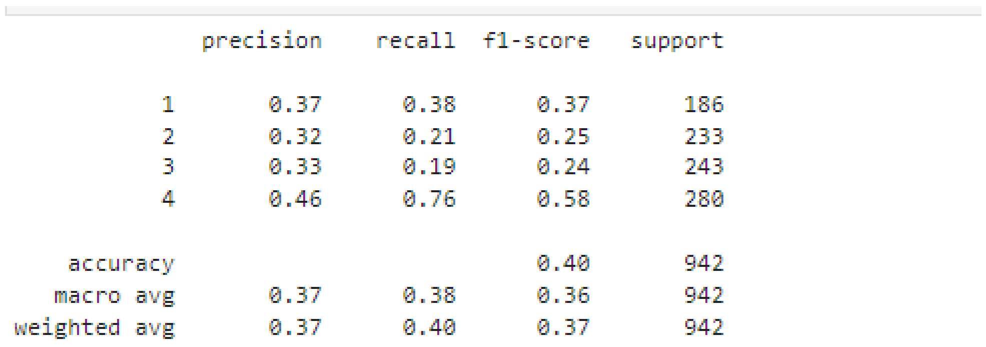


```
              precision    recall  f1-score   support

           1       0.40      0.34      0.37       186
           2       0.35      0.31      0.33       233
           3       0.31      0.28      0.29       243
           4       0.53      0.67      0.59       280

    accuracy                           0.42       942
   macro avg       0.40      0.40      0.40       942
weighted avg       0.40      0.42      0.41       942
```

Support Vector Machine - Linear

Next, we ran a Support Vector Machine – Linear classifier on the data set, with max iterations set to 10,000 for the algorithm to converge. The classification report shows that performance is a little worse than that of the KNN:

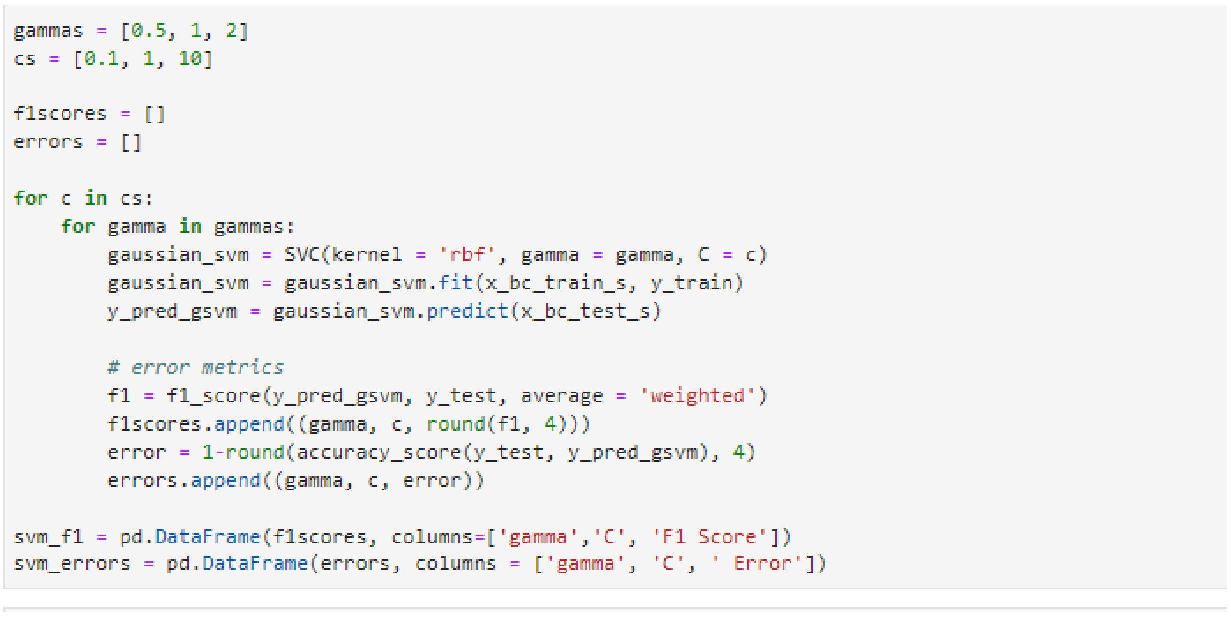
- Similar performance at classes 1 and 4 (high and low salary grades)
- Performance worse in classes 2 and 3



```
              precision    recall  f1-score   support

           1       0.37      0.38      0.37       186
           2       0.32      0.21      0.25       233
           3       0.33      0.19      0.24       243
           4       0.46      0.76      0.58       280

    accuracy                           0.40       942
   macro avg       0.37      0.38      0.36       942
weighted avg       0.37      0.40      0.37       942
```

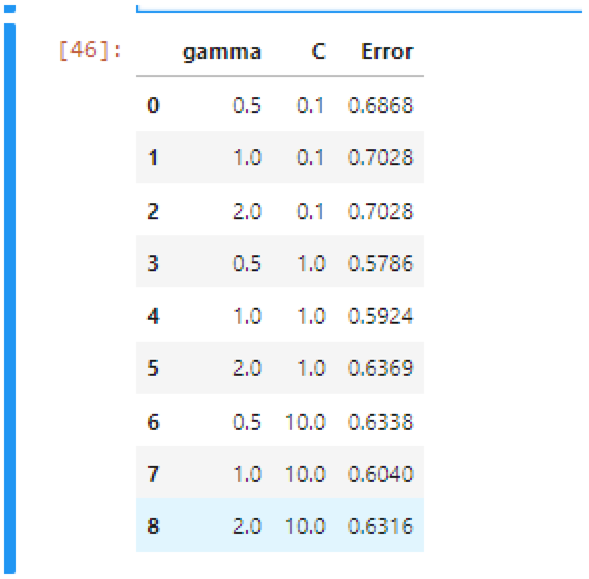
Support Vector Machine – RBF Kernel

Next, we ran a kernel SVM classifier, iterating over hyperparameters to get the best classifier. Hyperparameters we varied in a for loop as follows:



```python
gammas = [0.5, 1, 2]
cs = [0.1, 1, 10]

f1scores = []
errors = []

for c in cs:
    for gamma in gammas:
        gaussian_svm = SVC(kernel = 'rbf', gamma = gamma, C = c)
        gaussian_svm = gaussian_svm.fit(x_bc_train_s, y_train)
        y_pred_gsvm = gaussian_svm.predict(x_bc_test_s)

        # error metrics
        f1 = f1_score(y_pred_gsvm, y_test, average = 'weighted')
        f1scores.append((gamma, c, round(f1, 4)))
        error = 1-round(accuracy_score(y_test, y_pred_gsvm), 4)
        errors.append((gamma, c, error))

svm_f1 = pd.DataFrame(f1scores, columns=['gamma','C', 'F1 Score'])
svm_errors = pd.DataFrame(errors, columns = ['gamma', 'C', ' Error'])
```

A review of the error terms shows that the best model was generated at a combination of gamma = 0.5, C = 1.0



[46]:

	gamma	C	Error
0	0.5	0.1	0.6868
1	1.0	0.1	0.7028
2	2.0	0.1	0.7028
3	0.5	1.0	0.5786
4	1.0	1.0	0.5924
5	2.0	1.0	0.6369
6	0.5	10.0	0.6338
7	1.0	10.0	0.6040
8	2.0	10.0	0.6316

Re-running the model with the best set of hyperparameters, we find that model performance is quite like that of the KNN classifier. See the below classification report:

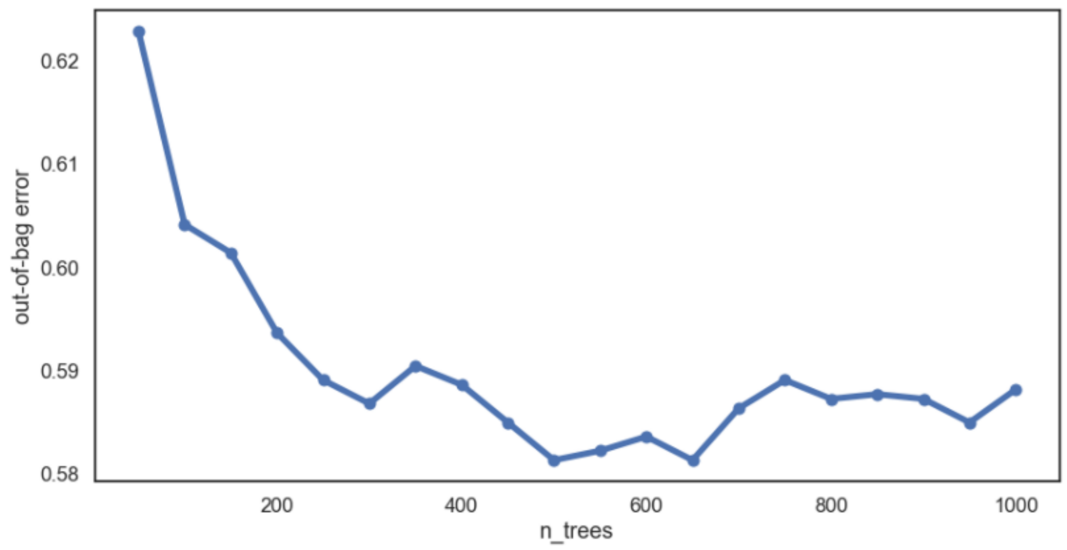
- The gaussian SVM is a little more accurate in salary class 3 and 4
- The KNN is a little ore accurate on classes 1 and 2

```
              precision    recall  f1-score   support

           1       0.44      0.29      0.35       186
           2       0.33      0.33      0.33       233
           3       0.32      0.34      0.33       243
           4       0.55      0.66      0.60       280

    accuracy                           0.42       942
   macro avg       0.41      0.40      0.40       942
weighted avg       0.42      0.42      0.41       942
```

Random Forest

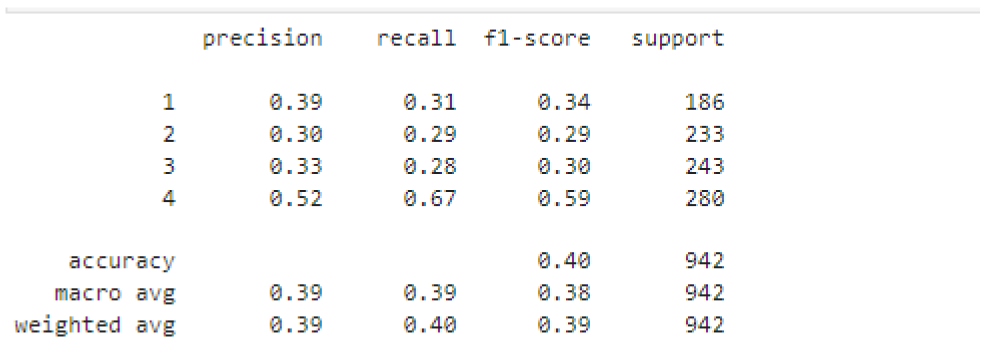
Finally, it was decided to attempt a tree-based model, to see if information theory did a better job of classifying than the currently best performing distance-based calculations.

The Random forest classifier was run, again in a loop to determine the optimal number of trees. The metric used to determine the 'best' solution was out of bag error (oob_error). Looking at the results, it appears the best model is one with 500 trees – again, with a relatively high out of bag error (a common theme across all the classification models)



Re-running the model at n_trees = 500, we find again similar performance to the KNN and SVM RBF Kernel. See the classification report below:

- The Random forest (information gain) did not provide any performance improvement over the KNN or Gaussian SVM



```
              precision    recall  f1-score   support

           1       0.39      0.31      0.34       186
           2       0.30      0.29      0.29       233
           3       0.33      0.28      0.30       243
           4       0.52      0.67      0.59       280

    accuracy                           0.40       942
   macro avg       0.39      0.39      0.38       942
weighted avg       0.39      0.40      0.39       942
```

<u>Key Findings</u>

As briefly mentioned during the review of each model, the algorithms do not do a great job of classifying player salary based on advanced statistics. This could be due to several reasons, including:

- Salary (by contract) is usually determined by past performance. Player's sign contracts based on past output, and to a smaller extent, the team's idea/projection of their future output. Depending on where the payer is in their contract term, their age, injury history, and changing team environment output can/cannot match salary outlay
- As a player's circumstances change (age, injuries) it may be more difficult to match previous statistical output/efficiency
- Changing Teams: how much of a player's statistical output is a function of the environment/system in which they were playing?
- Advanced vs Counting Stats: there are many bench players, who's advanced statistics compare favorably in small sample sizes, which could throw off salary predictions (the look like advanced stat stars in small sample but are paid like the role players they are). Typically, in larger samples, their advanced stats regress to the level at which they're paid
- The performance of many players in the NBA is essentially interchangeable. This can be seen in the classification report - the higher the salary class (higher end/Star Players) the better job the algorithm does of predicting salary class.
- Salary Appreciation/Inflation: over several years, and successive collective bargaining agreements (CBAs), the average salary of the NBA player has exploded – highly paid players in 2002 – 03 are average paid players in 2017 – 18. This brings up a couple of issues
    - Due to the relatively small number of records, it is not a good idea to further shrink the dataset
    - Perhaps it is instructive to come up with a salary metric that compares the individual salary to the min/mx/average salary for a given season, as the set point between performance and salary is becoming skewed by ever changing market conditions

Based on this, it may be worthwhile to add an age/experience/injury history/team performance component to the analysis. Additionally, some normalization of the salary term may be instructive, due to changing salary landscapes (although the performance metrics do not change). It appears that the analysis uncovers a little insight, though it is incomplete.

Further Analysis

As described above, the analysis is quite complex, as there are several moving parts. The analysis shows a little insight, but in general does a poor job of classifying salary based on advanced statistical outputs.

Some of the points made above that will likely do the most for adding insight would be:

- Determining where a player is in their contract term. This way, we can determine how far away the player is (in years) from the season/performance that got them the certain contract/salary
- Coming up with a salary metric that compares a player's salary to min/max/average – likely a yearly percentile salary term. This way, performance vs salary can be normalized and not evaluated across a data set that appreciates heavily in salary, with no real change in performance metrics

Some challenges that may be encountered in trying to improve this analysis:

- Data volume: there are only so many contracts (in our current data set, we're looking at a little over 200 records per year), so creating percentiles for salary each year may be difficult with so few records
- Contract terms: contract data and terms only extend back so far – finding a data set that summarizes when contracts were signed, with relatively clean terms has thus far been elusive, at least from publicly available data
- Merging advanced statistics with salary data to grow the dataset: it is difficult to find publicly available datasets that are larger than those currently offered up