NFL Contract Prediction

Presented By: Julian Ibarra

Date: 29, October 2020

Problem Statement:

- To **classify** if the player will play under contract in each remaining year of his deal.
- 1 -> Yes, Success, etc.; 0-> No, Failure, etc.
- Given a set of new predictor variables, the goal is to make predictions using the target variable (is_dead) as 1 -> 'Contract Is Dead' or 0 -> 'Contract Is Not Dead'.
- The metric that being used to choose the best model will is 'False Negative Rate'. (The predictor and target variables will be explained later).

Data Engineering:

- The data was taken from Spotrac and formatted into Excel sheets for analysis.
- The collected data is from the 2019 season onward and separated into two separate sheets: one for training and validation and another for predictions.
- The data set consists of 2997 observations and 44 features on the test/validation set and 618 observations and 44 features on the prediction set.
- Assumptions: features which we’re not relevant to the goal were removed.

Data Engineering:

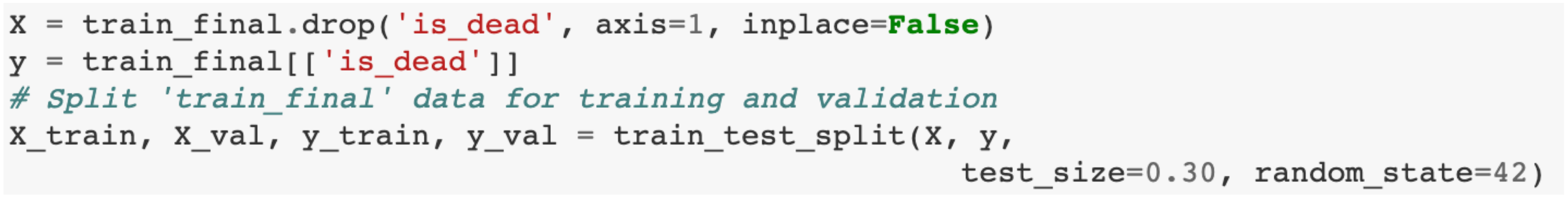
Data Cleaning and Transformation:

1. Redundant variables have been dropped.
2. String values have been formatted to integers.
3. Categorical values have been transformed to numerical values.
4. Used winsorizing to limit extreme values and reduce effect of outliers.
5. Filled in ‘NAN values with 0.
6. All the numerical values have been scaled to a range between -1 and 1.
   Used MinMaxScaler() function. Scaling allows features to be weighed equally.

After cleansing and transforming the data, correlation and an in depth reference to Spotrac’s data dictionary is used to reduce both datasets to 34 features. If feature correlation was above .45 than it’s dropped.

Splitting Data:

- Train_Test_Split() function on the test data
- ‘TRAIN’ dataset = 70% Training, 30% Validation, used entire ‘TO PREDICT’ set to run predictions.



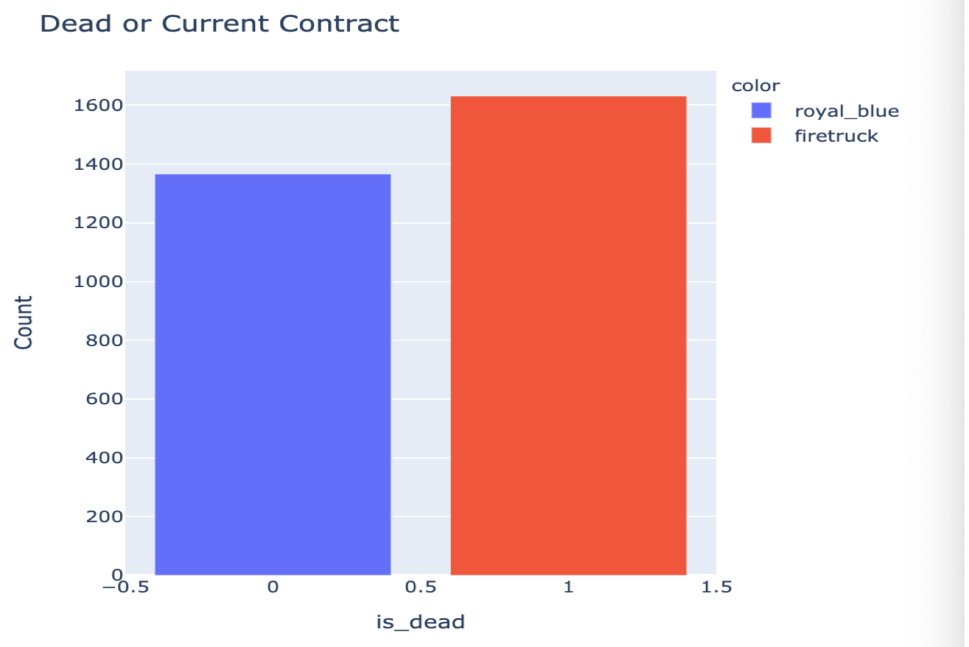
```
X = train_final.drop('is_dead', axis=1, inplace=False)
y = train_final[['is_dead']]
# Split 'train_final' data for training and validation
X_train, X_val, y_train, y_val = train_test_split(X, y,
                                                    test_size=0.30, random_state=42)
```

Recursive Feature Elimination:

- On the above 34 features, Recursive Feature Elimination (RFE) with cross validation was used to get the best 10 features.
- Recursive feature elimination is basically a backward selection of the predictors. This technique begins by building a model on the entire set of predictors and computing an importance score for each predictor. The least important predictor(s) are then removed, the model is re-built, and importance scores are computed again.

Target Variable:

- The target variable is ‘is_dead’ which shows if the contract is dead or not.
- 0 = ‘Contract is not dead’; 1 = ‘Contract is dead/Player played through remainder of his contract’
- Because the target variable is a categorical dependent variable, supervised learning/classification modeling will be performed.



Observations: ***(training set only!)***

- Approximately 1390 players did not continue to play through their remaining contract.
- Approximatley 1600 players did play through their remaining contract.
- The dataset is unbalanced where one category outnumbers the other.

```
# Plot 'is_dead' using final training data
sns.countplot(train_final['is_dead'])
```

Models Applied and Motivation:

- **Logistic Regression:** With logistic regression, the outputs have a nice probabilistic interpretation, and the algorithm can be regularized to avoid overfitting.
- **Support Vector Machine Model:** SVM is effective for this problem because of the hyperplane is found, the data other than the support vectors become redundant. This means that small changes to the data cannot greatly affect the hyperplane and SVM. It also generalizes really well.
- **KNN:** KNN works by finding the distances between a query and all the examples in the data, selecting the specified number examples (K) closest to the query, then votes for the most frequent label or averages the labels.

Grid Search:

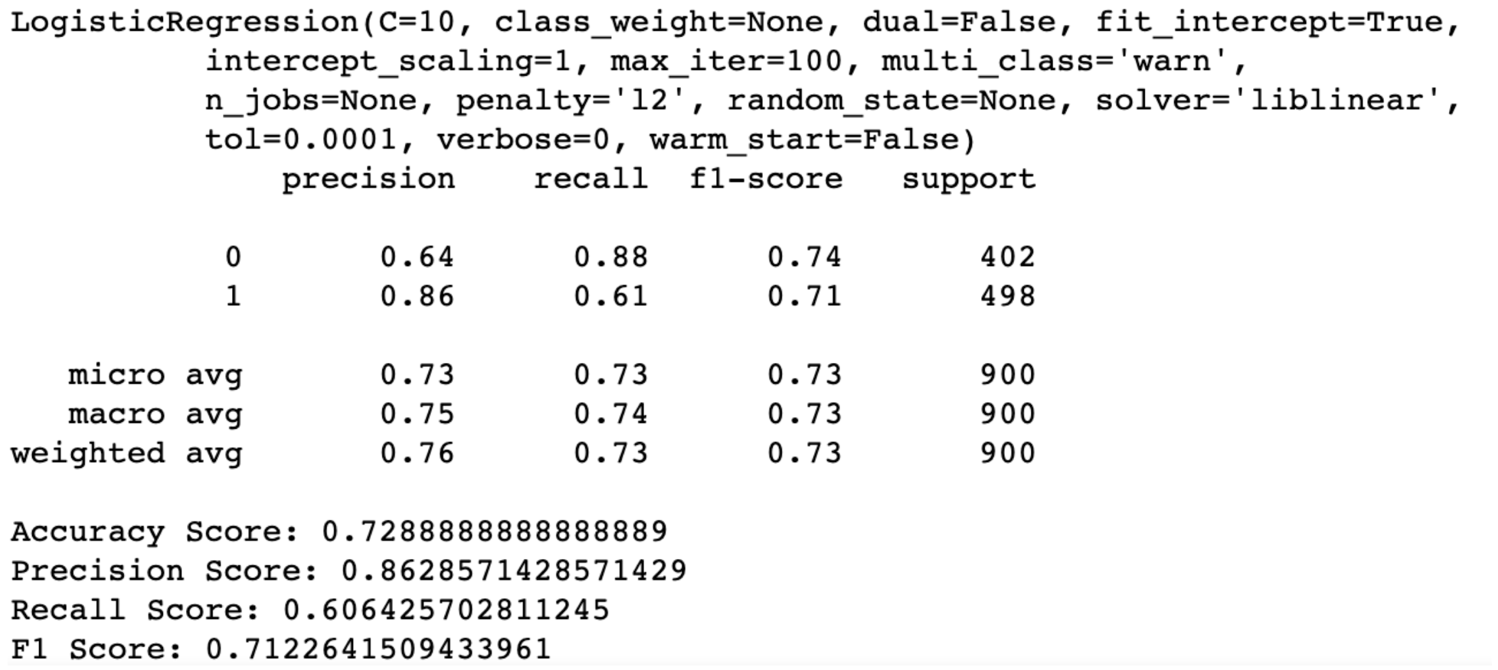
- Grid Search is a tuning technique that attempts to compute the optimum values of hyperparameters. It is an exhaustive search that is performed on a the specific parameter values of a model. The model is also known as an estimator.

Approach:

- I will create several classifiers and run them against the training dataset using GridSearch to find optimum hyperparameters.

Logistic Regression using Grid Search:

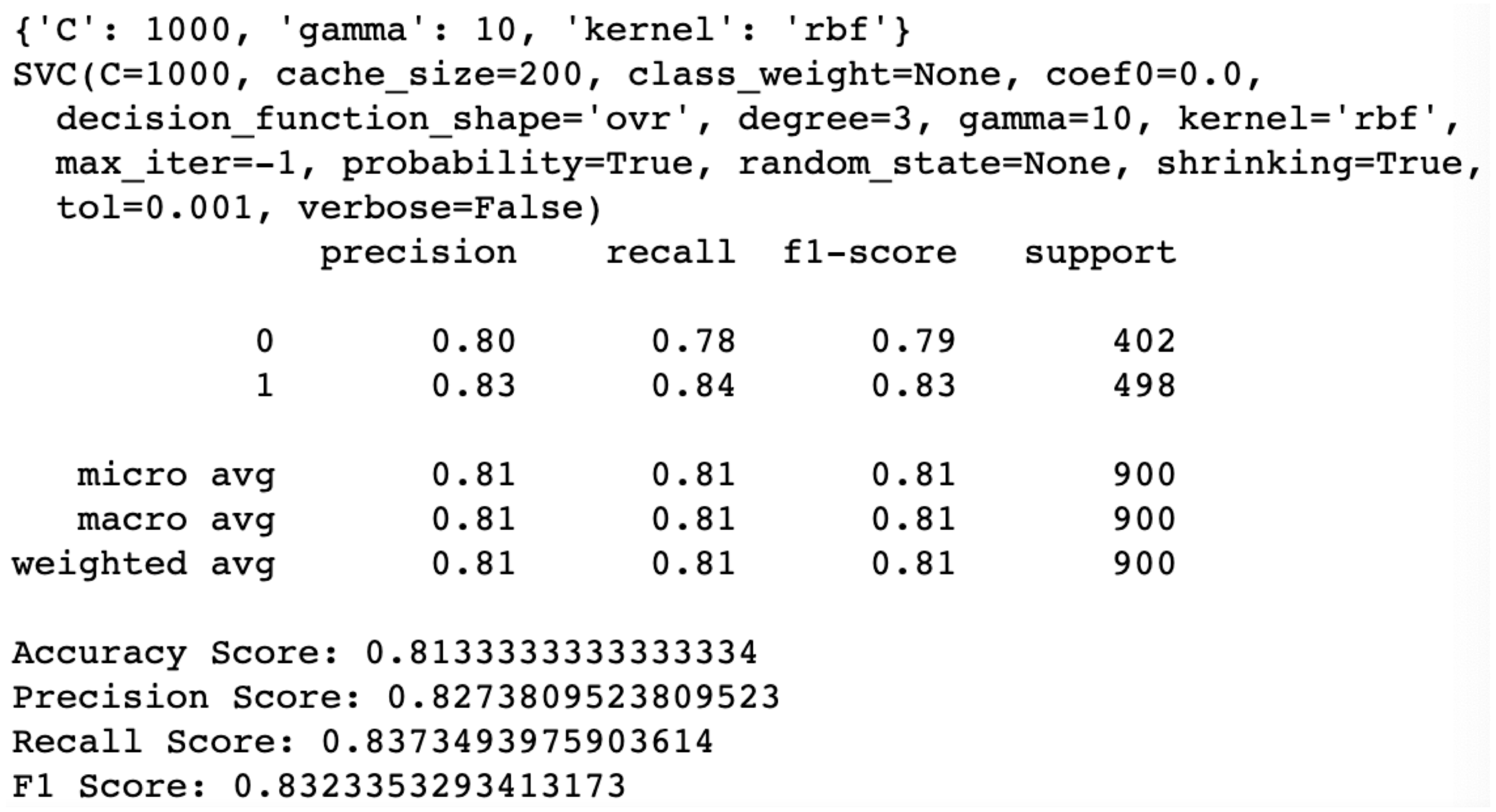
- My code printed out the best hyperparameters:
- Regularization Parameter L2: Ridge Regression
- Trade of Parameter C: 10 – Strength of regularization parameter



```
LogisticRegression(C=10, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='liblinear',
          tol=0.0001, verbose=0, warm_start=False)
              precision    recall  f1-score   support

           0       0.64      0.88      0.74       402
           1       0.86      0.61      0.71       498

   micro avg       0.73      0.73      0.73       900
   macro avg       0.75      0.74      0.73       900
weighted avg       0.76      0.73      0.73       900

Accuracy Score: 0.7288888888888889
Precision Score: 0.8628571428571429
Recall Score: 0.606425702811245
F1 Score: 0.7122641509433961
```

SVM using Grid Search:

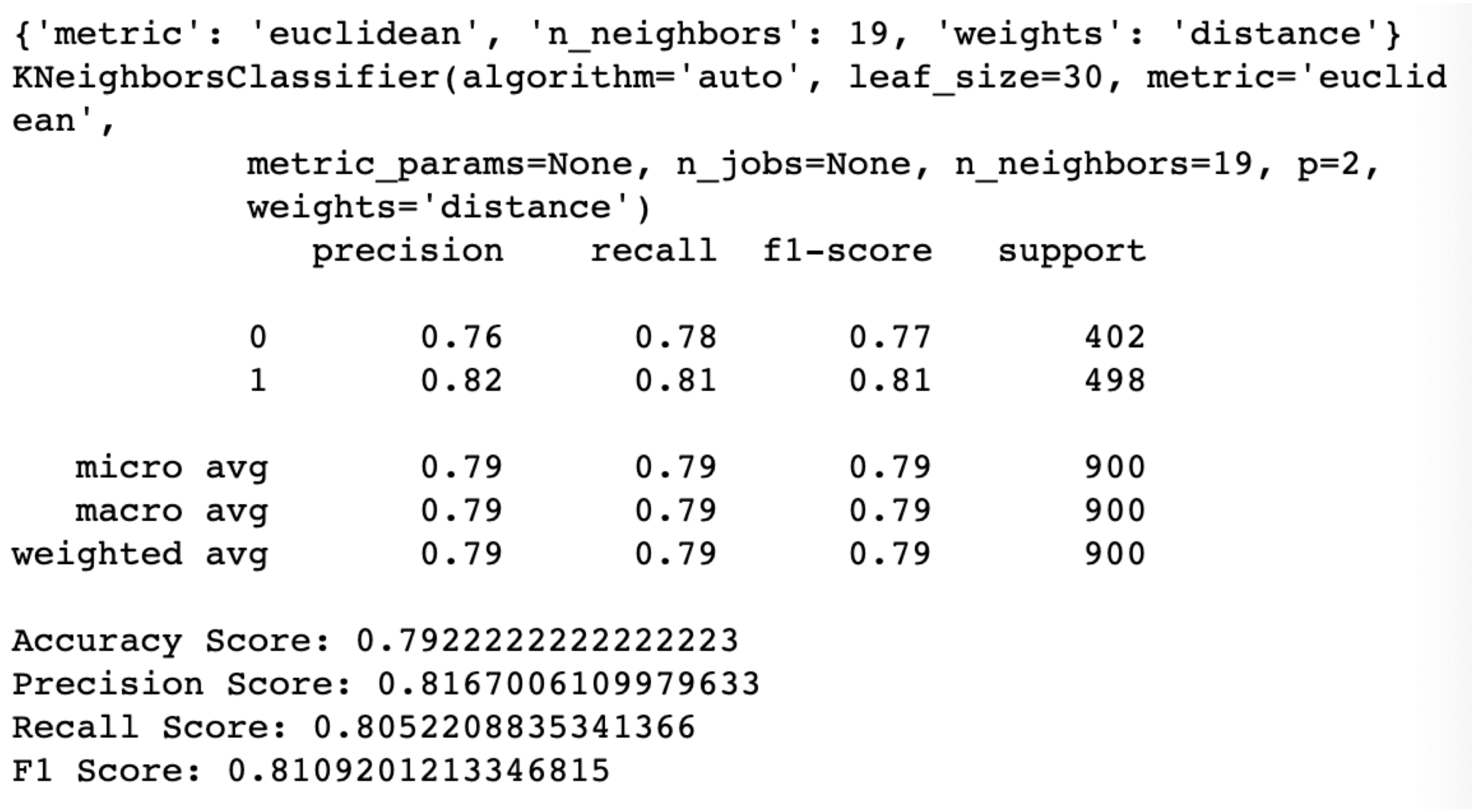
- My code printed out the best hyperparameters :



```
{'C': 1000, 'gamma': 10, 'kernel': 'rbf'}
SVC(C=1000, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=10, kernel='rbf',
  max_iter=-1, probability=True, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
              precision    recall  f1-score   support

           0       0.80      0.78      0.79       402
           1       0.83      0.84      0.83       498

   micro avg       0.81      0.81      0.81       900
   macro avg       0.81      0.81      0.81       900
weighted avg       0.81      0.81      0.81       900

Accuracy Score: 0.8133333333333334
Precision Score: 0.8273809523809523
Recall Score: 0.8373493975903614
F1 Score: 0.8323353293413173
```

KNN using Grid Search:

- My code printed out the best hyperparameters :

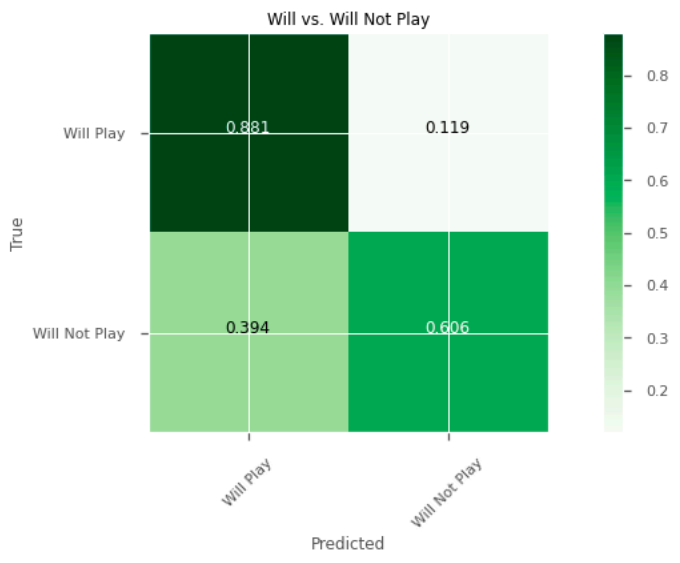


```
{'metric': 'euclidean', 'n_neighbors': 19, 'weights': 'distance'}
KNeighborsClassifier(algorithm='auto', leaf_size=30, metric='euclid
ean',
           metric_params=None, n_jobs=None, n_neighbors=19, p=2,
           weights='distance')
              precision    recall  f1-score   support

           0       0.76      0.78      0.77       402
           1       0.82      0.81      0.81       498

   micro avg       0.79      0.79      0.79       900
   macro avg       0.79      0.79      0.79       900
weighted avg       0.79      0.79      0.79       900

Accuracy Score: 0.7922222222222223
Precision Score: 0.8167006109979633
Recall Score: 0.8052208835341366
F1 Score: 0.8109201213346815
```

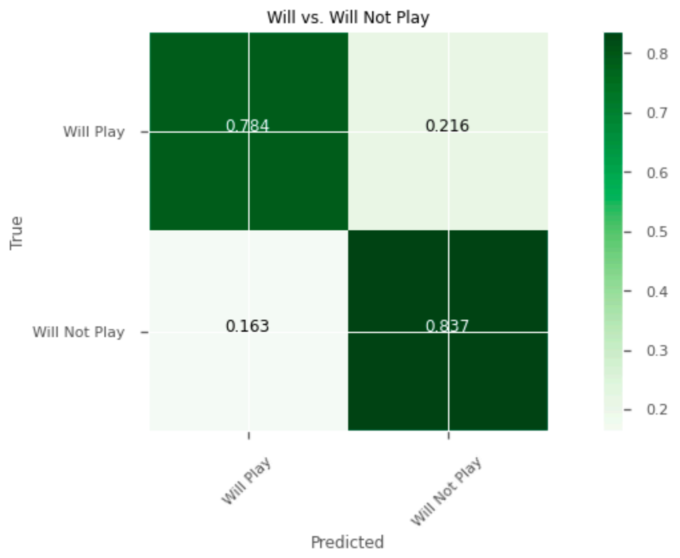
Evaluation Approach

- No universal best measurement approach
- For my prediction, False Positive Rate is the best metric to evaluate the model.
- The lower number of False positives, the better the model is.
- False Positive: “A player will player under contract even though he won’t”.

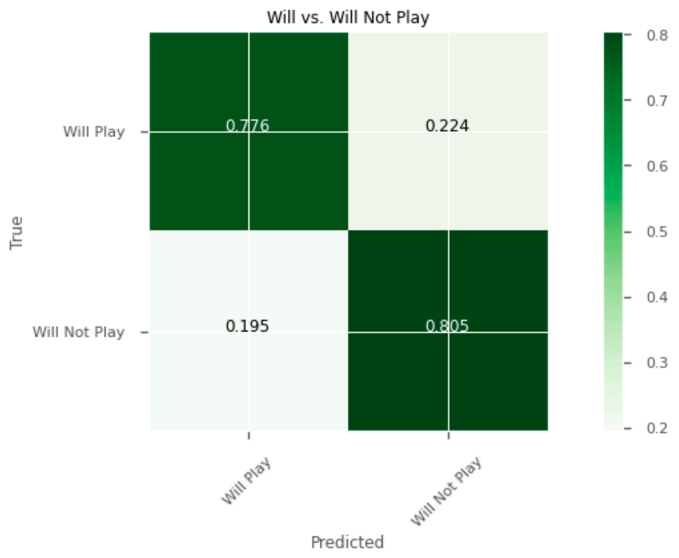
Logistic Regression



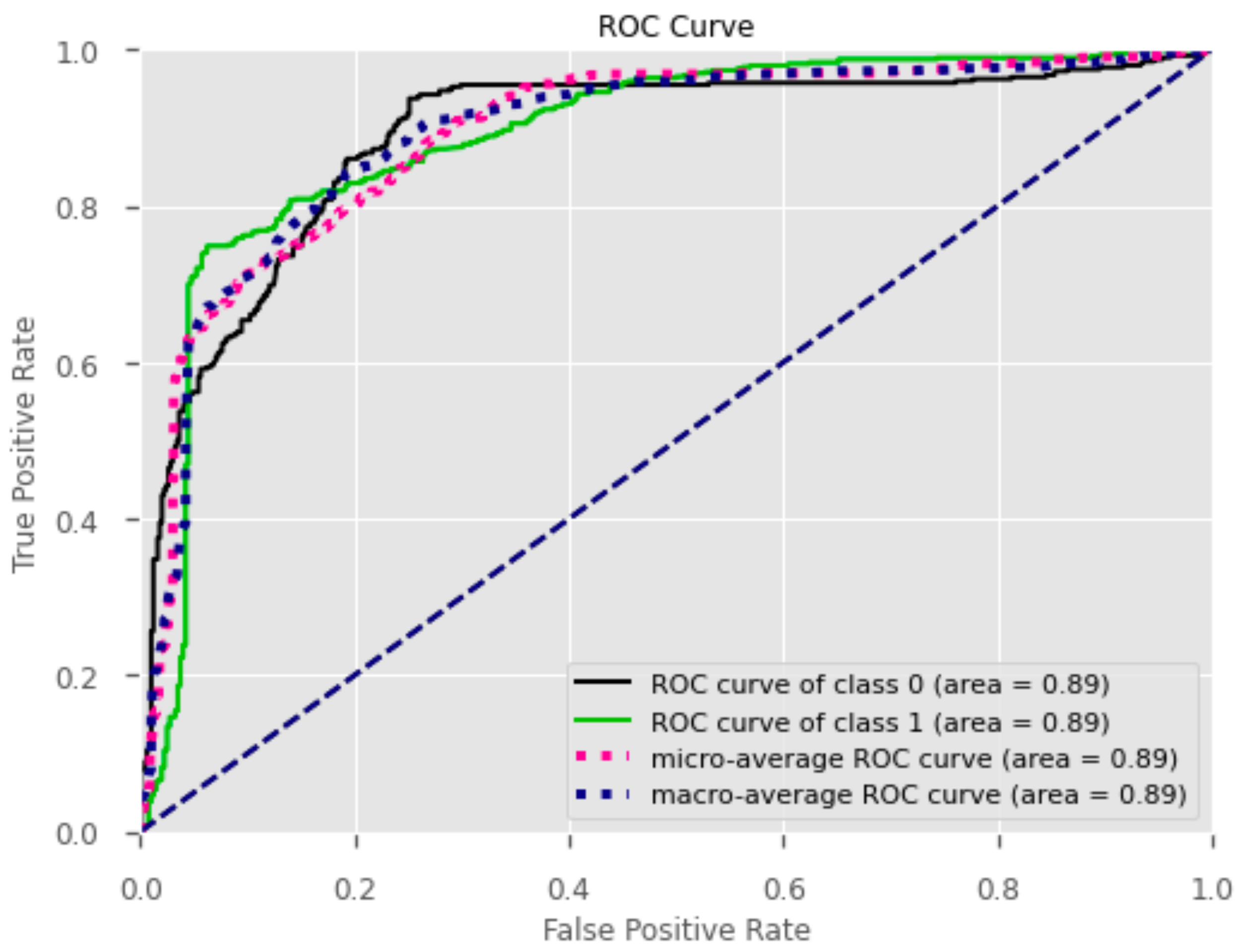
Support Vector Machine



K Nearest Neighbors



Receiver Operating Characteristic Curve:



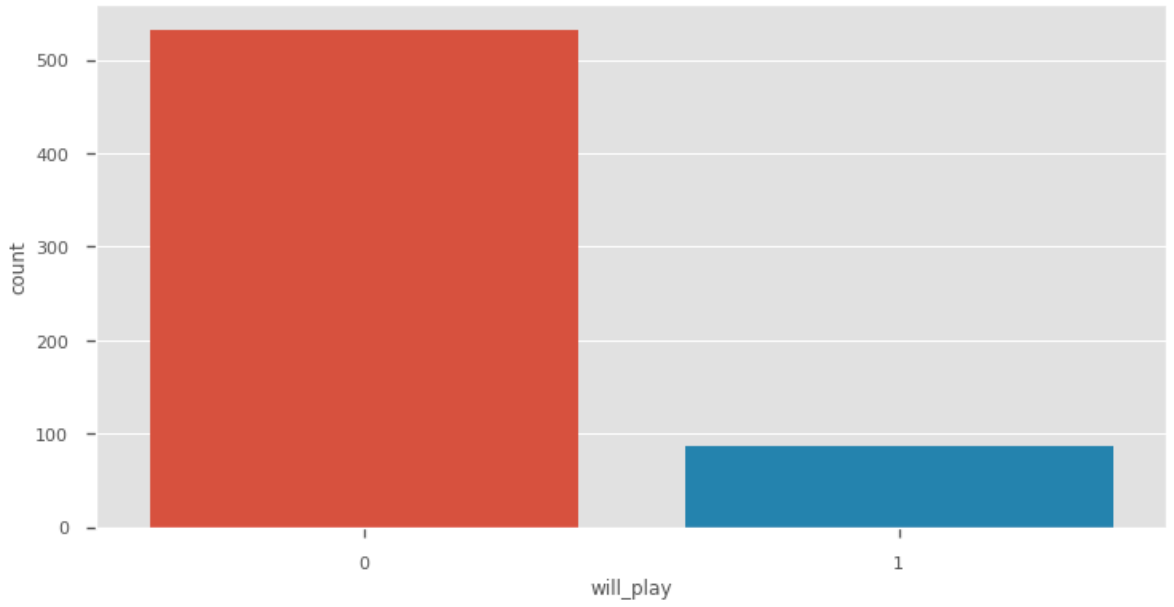
- AUC is the probability that a random positive example is positioned to the right of a random negative example.
- AUC ranges in value from 0 to 1.
- A model whose predictions are 100% wrong has an AUC of 0.0; one whose predictions are 100% correct has an AUC of 1.0.
- he implicit goal of AUC is to deal with situations where you have a very skewed sample distribution, and don't want to overfit to a single class.

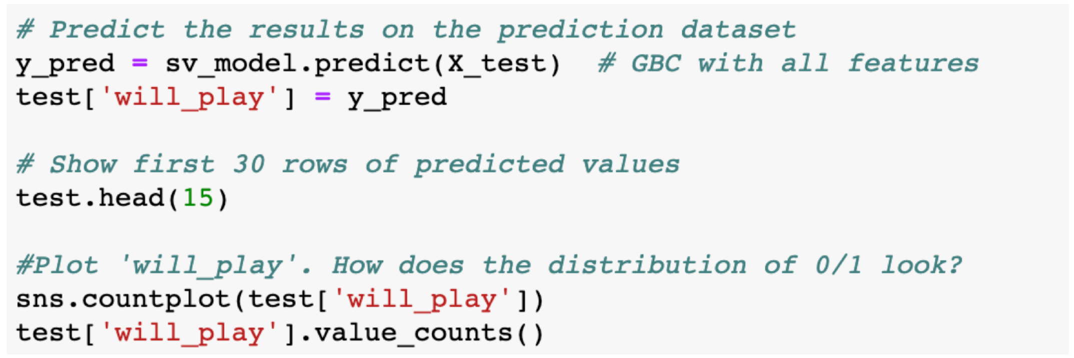
Model Performance Measurement:

- Precision – high cost of false positive, low cost of false negative
- Its an issue for a player to thought to be playing under contract even though he wont – cause a team to acquire a dead cap hit - therefore we want to reduce the chances of this happening.
- Logistic Regression with Grid Search: **0**: 86%, **1**:64%, FP: 11.9%
- Support Vector Machine with Grid Search: **0**: 80%, **1**:83%, FP: 21.6%
- KNN with Grid Search: **0**: 79%, **1**:81%, FP: 22.4%

Final Prediction on Test Set:

- Predicted the results on the test dataset using the Support Vector Machine model.
- Created a column ‘Will_Play’ which displays a 1 = ‘Will Play through Contract’ and 0 = ‘Will Not Play through Contract’





```
# Predict the results on the prediction dataset
y_pred = sv_model.predict(X_test)  # GBC with all features
test['will_play'] = y_pred

# Show first 30 rows of predicted values
test.head(15)

#Plot 'will_play'. How does the distribution of 0/1 look?
sns.countplot(test['will_play'])
test['will_play'].value_counts()
```

Future Applications:

- Implement a learning curve to identify the prediction accuracy/error vs. the size of my training set. It would basically allow me to get a better idea of how well the model predicts my target value as I increase the number of instances I use to train it.
- Utilize additional classification models, as only three were used for this assignment. (i.e: Naive Bayes, Decision Tree, Random Forest etc.).
- Use the time to perform more elegant data cleansing and transformation.