ATP Tennis Outcome Prediction using Machine Learning

Aaron Palmer, Aidan Reeve, Brody Galloway

2023-04-11

Introduction

This report applies machine learning methods to predict the outcomes of tennis matches with the goal of determining how predictable tennis matches are and subsequently, what factors have the largest impact on this predictability. The report uses Association of Tennis Professionals (ATP) match data collected throughout 2000-2023 with the data up until March 2023. Using the raw performance and player stats, features were created to better measure performance such as aces to double faults, Aces per serving point, and more. The report will use these features along with player statistics such as age, height, and dominant hand to predict the outcome of the match. The report will compare the results of different machine learning models and address whether the predictability of the outcome is beyond that of a traditional method. Finally, after optimizing each models' parameters and hyperparameters, this report will highlight what is deemed to be the best model for the prediction and discuss the findings.

Motivation

As the three of us are all avid sports fans we have always been fascinated what it takes to become a champion. This has always resulted in lively conversations that have taken over our living rooms during major sports events. This has evolved into a deep interest in sports analytics and breaking it down by the statistics. When this project came up and a discussion regarding a topic began, we discussed doing a research paper on various financial topics but finally came to the decision to find a topic within the realm of sports and measuring what it takes to win. In a world where sports analytics have come to the forefront of discussion for many sports, we saw this as an opportunity to apply our knowledge of machine learning to this topic. While conducting research we came across an extremely in-depth data set with almost thirty years of data and close to fifty variables for ATP tennis matches. This was obviously very intriguing and ended up being perfect considering we have all been tennis fans for the past few years.

The application of machine learning to the game of tennis is a topic that has been covered before but we believed with the data set we acquired that we would be able to create a unique and improved model for predicting ATP matches. The previous academic research focused on historical data to predict future matches with the goal of creating sport betting models. These academic papers employed the use of neural network to attempt to predict winners with varying results. We believe that we would be able to provide a deeper analysis and research into the underlying variables that affect the outcome of these matches. Tennis is an interesting sport when it comes to statistics because it contains a wide range of variables which can be used. This sparked out interest as we began to go through the logistics of how we could apply the machine learning models it became more apparent that this idea was feasible and would work well.

As machine learning researchers and avid sports fans, we are very excited to dive into this topic creating a unique mix of sports analytics and machine learning. We have high expectations that the models will be accurate and interpretable to be able to draw valuable insights into underlying variables that effect match outcomes. Ultimately, we believe that this research will contribute to new techniques and applications of machine learning and sports analytics in tennis.

Data

Our ATP tennis data was sourced from Jeff Sackman's GitHub repository, the repo contains all data from ATP tennis matches from the late 1960s to today. For our project we used the tennis match data from 2000 until 04/04/2023, when the data was pulled.

There are more comments relating to data creation and pre-processing in line with the code in the Code section of the ToC.

Variable Definitions

Feature Name	Description
Ace/df	ratio of player aces to player double faults
Svpt/game	ratio of player serve points to serve games
Bpsaved/bpfaced	ratio of breaking points saved to breaking points faced
Ace/svpt	ratio of player aces to serving points
1stWon/svpt	ratio of 1st serve points to total serve points
2ndWon/svpt	ratio of 2st serve points to total serve points
outcome	win/loss, 1 for win, 0 for loss

Feature Name	Description
age	player age in years
ht	player height in cm
hand	player dominant hand, right/left, 0 for left, 1 for right

Summary Statistics

The table of summary statistics can be referenced in the Code -> Summary Statistics section.

Each feature has around 125k samples (around 63k matches each with a winner and loser). The rank_points represents the players rank points at the start of the tournament week, the values range from 1 to ~17,000 with around half of the values between 1 and 835 and an average of ~1300. The ace/df feature has an average of 2.87 and spans between 0 and 49, with 75% of values between 0 and 4. The svpt/game has an average of 6.39 and varies from 0 to 88. There are some outliers in this section as the 25th-75th percentiles are 5.8 – 6.88. The bpsaved/bpfaced ratios range between 0 and 1 with an average of 0.58. Ace/svpt has a mean of 0.08 and ranges from 0 to 0.81, this feature also has some outliers as the 25th to 75th percentiles span from 0.03 to 0.11. Both the 1stWon and 2ndWon per serving point features range from 0 to 1, the 1stWon/svpt has a mean of 0.43 and the 25th-75th percentiles span from 0.38 to 0.49. The 2ndWon/svpt has an average of 0.2 and 25th-75th percentiles span from 0.16 to 0.24.

We can also see the histogram of our features to get a glimpse into the distribution of our variables. Our created variables have solid distributions and should be appropriate for the problem we're trying to solve.

Methodology

Using the following statistical models, we can attempt to predict the winner of each match and compare against our testing set. We will discuss the process and mechanisms that the models use to maximize each objective function. Each model attempts to minimize their Mean Squared Error (MSE) which in turn increases accuracy.

As we set up our models, we used different Cross Validation (CV) techniques to transform our data into useable information. For each regularized model we used K-Fold CV to create the training and test sets. For our models we used k=5 folds to find a balance between bias and variance while keeping our computational expenses as low as possible. K-fold CV works by dividing the full data set into k-number of random sections, from there one section is used as the test set while the rest is the training set. The algorithm repeats this process k-number of times until every section has had a chance to be the test set. Similarly, to K-Fold CV for all our regularized models we used Grid Search CV to create more specific models. Grid Search

CV uses a grid of hyperparameter values to for each model then evaluates the performance to determine the best combination. This will allow us to test a large range of hyperparameters across our models such as c-value, neighbour values, and n-estimator. Once the algorithm has determined the best combination of hyperparameters it is selected and set as the configuration the model will use. This becomes incredibly effective when we are looking at a vast range hyperparameters; but can lead to overfitting the hyperparameters in the test set.

Following this section, the subsections below will explain the statistical models we used in our report. For each of the models we were expecting to have generally good results considering the sheer large amount of data. When you are able to have such a vast amount of data it offers every model a significant number of observations to train on and improve. In addition, our model contained feature engineered variables that contained ratios from two observations in the original dataset. This provided a more specific level of analysis and opened the door for us to achieve the most accurate results possible.

In the code section we discuss the methodology in how we acquired, cleaned, created sets, and processed the data into useable information. This will as well include many comment that explain the process of creating the models and an overview of how to interpret our data.

Unregularized Logistic Regression

We will begin with the most popular classification model that we will employ for this report. This model predicts a categorical outcome based on our independent statistical variables by fitting a curve to the data which in turn will separate the two classes of dependent variables, in our case, win or lose. We must note that this model does not have an additional penalty for including more estimators which can cause over-fitting of the model. In the following sections we will look at models that apply a penalty for every additional estimator.

Ridge

Our second model used is our first that is a penalized model, Ridge Classification. This model is an extension of logistic regression but uses a defense system to prevent overfitting and improve stability. Ridge uses L2 regularization by adding a penalty term to the loss function for each new parameter used in the model. These penalties cause the coefficient estimates to decrease towards each other (grouping effect) and zero but never arrive there which causes the model to have increased bias and decreased variance.

Lasso

The third model is our second penalized model, Lasso Classification. Similar to Ridge this model is a variant of logistic regression with a defense system for overfitting. In this case the penalty term is applied to the loss function for each new parameter; the nature of these penalties is that they are the absolute value of the coefficients. These penalties cause the coefficient estimates to decrease and can reduce multiple of them to zero. This creates a more simplistic model with less features (feature selection); in turn this can improve the interpretability of the results and similar to ridge increasing bias while decreasing variance.

Elastic Net

Next, we can look at Elastic Net which is a combination of the two previous models. Elastic Net Classification effectively combines the feature selection penalty from Lasso and the grouping effect features from the penalty in Ridge. This creates a model that can handle many correlated features better than the two prior models; as well as making it more flexible for large data sets similar to ours. Again, similarly to the previous models both penalties included lead our model to have increased bias and decreased variance.

K-Nearest Neighbours

Our next model, K-Nearest Neighbours, is a non-parametric method of classification that identifies the K-points in the training data closest to any given observation point. In our model the value of K is a hyperparameter that can affect the performance and bias; the smaller the value of K can lead to higher variance while a larger value of K can lead to lower variance. From there it creates different groups based on the distance from an observation point and calculates conditional probability. This provides insights into trends of our data by measuring how different sizes of data effect the accuracy.

Random Forest

Random Forest is our first model that uses multiple decision trees which improves accuracy and generalization in the model. In the classification method each decision tree is built using a random subset of both features and training data. This method reduces overfitting and increases the diversity among the trees. With each new observation Random Forest passes it through all the decision trees and assigns it to the section most frequently predicted like the observation. Finally, the output prediction is a combination of every prediction from each decision tree. This method is very helpful for us as its effective for complex data with non-linear relationships between the variables.

Gradient Boosted Trees

Moving on to our next model, similar to Random Forest, Gradient Boosted Trees Classification uses multiple decision trees which improves accuracy and generalization in the model. In this method the model creates decision tress sequentially and each time attempts to correct the errors from the previous tree. For each new observation the model processes it through each decision tree and combines each prediction using weighted averaging. Similar to Random Forest, this model is very helpful for complex data with non-linear relationships between the variables.

XGBoost

Extreme Gradient Boosting (XGBoost) is a variant of Gradient Boosted Trees which leverages a more optimized implementation strategy. Similar to Gradient Boosted Trees the model creates decision tress sequentially and each time attempts to correct the errors from the previous tree. The three main differentiating features are that XGBoost uses L1 and L2 regulation penalties like the Elastic Net model, it employs tree pruning which removes unnecessary tree branches, and missing value handling which means that the model can work with incomplete datasets. Similar to the past two models, this model is very helpful for complex data with non-linear relationships between the variables.

Model Assessment and Selection

As mentioned previously, we will assess the predictability of ATP tennis matches using a selection of models including: Unregularized Logistic Regression, Ridge, Lasso, ElasticNet, PCR, KNN, Random Forest, Gradient Boosted Trees and XGBoost. As you will see, all models performed almost identical predictive accuracy. However, this is a good sign as it shows that regardless of the model, we are getting consistent and reliable results.

Unregularized Logistic Regression

This model has the benefit of being simple and easy to understand. In this model, the logistic function is used to transform the linear combination into a probability value. The coefficients can be directly interpreted as the effect of each input feature on the predicted probability. The main risk with this model is overfitting when dealing with high-dimensional datasets. This model performed with a weighted average precision, recall and F1 score of 0.80.

Ridge

The Ridge Classifier is a linear classification model that uses L2 regularization to prevent overfitting. The L2 term penalizes large weights and encourages smaller ones to improve generalization performance. The most accurate parameter was C=1. This model performed with a weighted average precision, recall and F1 score of 0.80.

Lasso

The Lasso Classifier is similar to the Ridge, with main difference being that Lasso uses L1 regularization. Meaning that the penalty term is added to the sum of absolute values of the coefficients as opposed to the sum of squared coefficients (L2). Another benefit to the Lasso is that is allows for the weights of some features to be exactly zero. The best performing parameter was C=10. This model performed with a weighted average precision, recall and F1 score of 0.80.

ElasticNet

The ElasticNet Classifier is a technique that combines both L1 and L2 penalties in order to overcome some of the limitations of using either Ridge or Lasso. The main advantage of this model is that it can handle highly correlated features well. The best performing parameters for ElasticNet was C=1 and an L1 ratio of 0.25. The model performed with a weighted average precision, recall, and F1 score of 0.80.

K-Nearest Neighbours

The KNN Classifier is an algorithm that works by finding the k nearest neighbours to a given data point in training set and using the majority class of the neighbours to predict the class or value of the new data point. The best performing number of neighbours was 170. With 170 nearest neighbours the model performed with a weighted average precision, recall and F1 score of 0.79.

Random Forest

The Random Forest Classifier is an algorithm that combines multiple decision trees to improve accuracy and reduce overfitting. The main advantage of Random Forest is it very accurate for noisy or missing data. Another advantage is that the model can provide information on feature importance, which is useful for data exploration.

The best performing hyperparameters for Random Forest were n_estimators = 500 and max_depth = 15. The n_estimators hyperparameter is used to control the number of decision trees to be included in the model. The max_depth hyperparameter controls the maximum depth of each decision tree, which is the number of levels between the root node and the leaf nodes.

The model performed with a weighted average precision, recall and F1 score of 0.80. The most important feature was the ratio of aces to serving points, with an importance slightly greater that 0.30.

Gradient Boosted Trees

Gradient Boosted Trees (GBT) Classifier is similar to Random Forest, with the main difference being that GBT builds trees sequentially, with new trees trained to correct the errors of the previous tree. GBT tend to be shallower than Random Forest as well.

The best performing hyperparameters for GBT were max_depth = 2, n_estimators = 500, subsample = 0.5. The hyperparameter subsample is used to control the fraction of training data that is used to fit each tree. Generally, a lower subsample (0.5) reduces overfitting thus raising bias and lowering variance, whereas a higher subsample (1.0) does the opposite. The learning rate is a hyperparameter that controls the magnitude of contribution of each individual tree to the final prediction. The learning rate is not included as a hyperparameter for the best estimator because it is used during training, not prediction. Similar to Random Forest, aces/svpt was the most important feature.

The model performed with a weighted average precision, recall and F1 score of 0.81.

XGBoost

The Extreme Gradient Boosting (XGBoost) classifier is a model build upon the GBT model to improve accuracy by providing multiple types of regularization (L1, L2, tree pruning, etc.).

The best performing hyperparameters were max_depth = 3 and n_estimators = 300. The results of the model were an average weighted precision, recall and F1 score of 0.81. The most important features were aces per serving points, serving points per game and 1st serve points to all serve points.

Discussion

With the growing importance of using analytics when trying to solve problems, machine learning has been at the forefront and will only continue to become more accessible as time goes on. We, as well as many experts believe that sports analytics as a skillset has become very important for coaches and analysts to bring out the best in their athletes and attain the best results possible. This has been displayed in leagues such as the NBA with the rise of 3-point shooting when previous generations relied on mid-range shots and close-range playmaking. We hope to see more professional sports adapting as new insights are brought to the table.

There are a couple of key limitations noted in our paper that could be expanded upon in further studies. We engineered our features based on our domain knowledge which although decent, would not be as strong as a professional player or coach. In future work a wider variety of features could be constructed, for example, making more ratios that included all features, or adding more features that were not included in our dataset. We also tried to balance our parameter tuning through GridSearchCV to achieve a good balance between accuracy and computational intensity. With more time and better processing power we could have tested more parameters although we expect the benefits of this to be low due to the similarity in accuracy scores between the different models. We also know that physical and mental condition plays a strong role in tennis outcomes. If experts were able to find a way to quantify a player's mental game we expect this would lead to greatly increased accuracy in prediction. Physical condition, like mental is very hard to quantify for a data-based model. Features could be constructed that consider the players recent match times and rest time as well as more in-depth stats like distance travelled during the match, or exertion levels. When comparing to traditional prediction methods as well as previous literature on the subject one key limitation is the lack of betting odds in our dataset. A more traditional approach uses betting odds, and implied probability to calculate an expected value for determining the winner. Overall, we thought our models did well, but more data would surely help rather than hinder the performance of the predictions.

Conclusion

In conclusion, we had lots of fun undertaking this project and using our knowledge gained in the classroom on a problem that we are passionate about. We tested a wide variety of models, from a simple logistic regression to XGBoost and found that all the models tested did an adequate job at predicting the outcome of tennis matches. Our models accuracy scores ranged from ~0.79 and ~0.81 over a wide range of complexity. This shows that our data, along with our constructed features are good predictors and that the model complexity and parameters used have less effect on the predictability. After reviewing results we found that the tree-based models did the best job in terms of accuracy of predictions with the XGBoost classifier slightly edging out the others. When looking at the feature importance graphs in the tree models we saw the top three remain consistent, with ace/svpt, 1stWon/svpt, and svpt/game being the three most important in all the models.

References

De Seranno, A. (2020). Predicting Tennis Matches Using Machine Learning. Master's Dissertation, (1-68)

Sackmann, J. (n.d.). Tennis databases, files, and algorithms [Data set]. Tennis Abstract. Retrieved from https://github.com/JeffSackmann/tennis_atp under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Code

Imports

```python
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import seaborn as sns
import time
import glob

plt.style.use('ggplot')
import warnings
warnings.filterwarnings('ignore')
```

Initial Data Creation

```python
# Get a list of all csv files in the current directory
csv_files = glob.glob('atp_matches_*.csv')

# Create a list to store the dataframes
```

```
dfs = []

# # Loop over the csv files
for file in csv_files:
    df = pd.read_csv(file)
    dfs.append(df)

# Concatenate the data frames in the list into a single master data frame
master_df = pd.concat(dfs, ignore_index=True)

# Save master_df as csv, then push to hub
master_df.to_csv('masterCSV.csv')
```

The steps above were done to create the ‘mastercsv’ file, a combination of each years matches file from 2000-2023 originally pulled from JeffSackmann’s GitHub(referenced at end). We then pushed this mastercsv to GitHub for easier access when working in Google Colab. We used a Google Colab notebook file when working on the code so that we could work simultaneously. We ran into much fewer troubles when trying to access the file from GitHub than when we were trying to access it locally.

Initial Cleaning

```
# Define the URL of the raw CSV file in your GitHub repository
# url = 'https://raw.githubusercontent.com/palmerac/tennis_atp_FIN4000/master/masterCSV.cs

# Read the CSV file into a pandas data frame
# df = pd.read_csv(url)
df = pd.read_csv('masterCSV.csv')
# print(df.info())

# Drop columns
cols_2_drop = list(range(0,12)) + list([14,22]) + list(range(16,20)) + list(range(24,28))
df = df.drop(df.columns[cols_2_drop], axis=1)

print(df.columns)
```

```
Index(['winner_hand', 'winner_ht', 'winner_age', 'loser_hand', 'loser_ht',
       'loser_age', 'w_ace', 'w_df', 'w_svpt', 'w_1stIn', 'w_1stWon',
       'w_2ndWon', 'w_SvGms', 'w_bpSaved', 'w_bpFaced', 'l_ace', 'l_df',
       'l_svpt', 'l_1stIn', 'l_1stWon', 'l_2ndWon', 'l_SvGms', 'l_bpSaved',
       'l_bpFaced', 'winner_rank', 'winner_rank_points', 'loser_rank',
       'loser_rank_points'],
      dtype='object')
```

For the final version of the project we referenced the mastercsv locally instead of via GitHub as we ran it locally instead of on Colab. Next, we removed the unnecessary features and checked what columns we now have remaining.

```
# Create seperate df's for the winner and loser stats, add an 'outcome' column that has 1

wnrs = df.filter(regex='^winner_|^w_').assign(outcome=1)
lsrs = df.filter(regex='^loser_|^l_').assign(outcome=0)

print(wnrs.columns)
print(lsrs.columns)
```

```
Index(['winner_hand', 'winner_ht', 'winner_age', 'w_ace', 'w_df', 'w_svpt',
       'w_1stIn', 'w_1stWon', 'w_2ndWon', 'w_SvGms', 'w_bpSaved', 'w_bpFaced',
       'winner_rank', 'winner_rank_points', 'outcome'],
      dtype='object')
Index(['loser_hand', 'loser_ht', 'loser_age', 'l_ace', 'l_df', 'l_svpt',
       'l_1stIn', 'l_1stWon', 'l_2ndWon', 'l_SvGms', 'l_bpSaved', 'l_bpFaced',
       'loser_rank', 'loser_rank_points', 'outcome'],
      dtype='object')
```

Next, we separated the winner's stats from the loser's stats and added an outcome column denoting either 1 for a win or 0 for a loss. We then check the columns again to ensure that both dataframes have the same columns.

```
# Remove winner/loser notation from columns and concat them back together to get combined
wnrs = wnrs.rename(columns=lambda x: x.replace('winner_', '').replace('w_', ''))
lsrs = lsrs.rename(columns=lambda x: x.replace('loser_', '').replace('l_', ''))

# Concatenate data frames
df_combined = pd.concat([wnrs, lsrs], axis=0, ignore_index=True).dropna()
df_combined.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 125143 entries, 0 to 139129
Data columns (total 15 columns):
 #   Column       Non-Null Count   Dtype
```

```
---  ------        --------------  -----
 0   hand          125143 non-null  object
 1   ht            125143 non-null  float64
 2   age           125143 non-null  float64
 3   ace           125143 non-null  float64
 4   df            125143 non-null  float64
 5   svpt          125143 non-null  float64
 6   1stIn         125143 non-null  float64
 7   1stWon        125143 non-null  float64
 8   2ndWon        125143 non-null  float64
 9   SvGms         125143 non-null  float64
 10  bpSaved       125143 non-null  float64
 11  bpFaced       125143 non-null  float64
 12  rank          125143 non-null  float64
 13  rank_points   125143 non-null  float64
 14  outcome       125143 non-null  int64
dtypes: float64(13), int64(1), object(1)
memory usage: 15.3+ MB
```

Finally, we rename the columns from both the winners and losers dataframes to remove the winner and loser notation. Now that the columns names are the same, we will concatenate them back together to get our combined dataframe.

Feature Engineering

Our main goal for feature engineering was to create ratios with the existing features we had. Since game and match length can vary greatly in tennis it is much more useful to standardize the features so that we can compare games that differ in length. The methodology behind this is similar to how financial ratios are used, for example using profit margin instead of straight profit.

```
df_combined['ace/df'] = df_combined['ace'] / df_combined['df']
df_combined['ace/df'] = np.where(np.isinf(df_combined['ace/df']),
                                 df_combined['ace'],
                                 df_combined['ace/df'])
df_combined['svpt/game'] = df_combined['svpt'] / df_combined['SvGms']
df_combined['svpt/game'] = np.where(np.isinf(df_combined['svpt/game']),
                                 df_combined['svpt'],
                                 df_combined['svpt/game'])

df_combined['bpsaved/bpfaced'] = df_combined['bpSaved'] / df_combined['bpFaced']
```

```
df_combined['ace/svpt'] = df_combined['ace'] / df_combined['svpt']
df_combined['1stWon/svpt'] = df_combined['1stWon'] / df_combined['svpt']
df_combined['2ndWon/svpt'] = df_combined['2ndWon'] / df_combined['svpt']
```

```
# Drop raw features and only keep engineered ones, as well as hand, height, ranking pts
df_final = df_combined.drop(df_combined.columns[2:13], axis=1)
df_final.info()
```

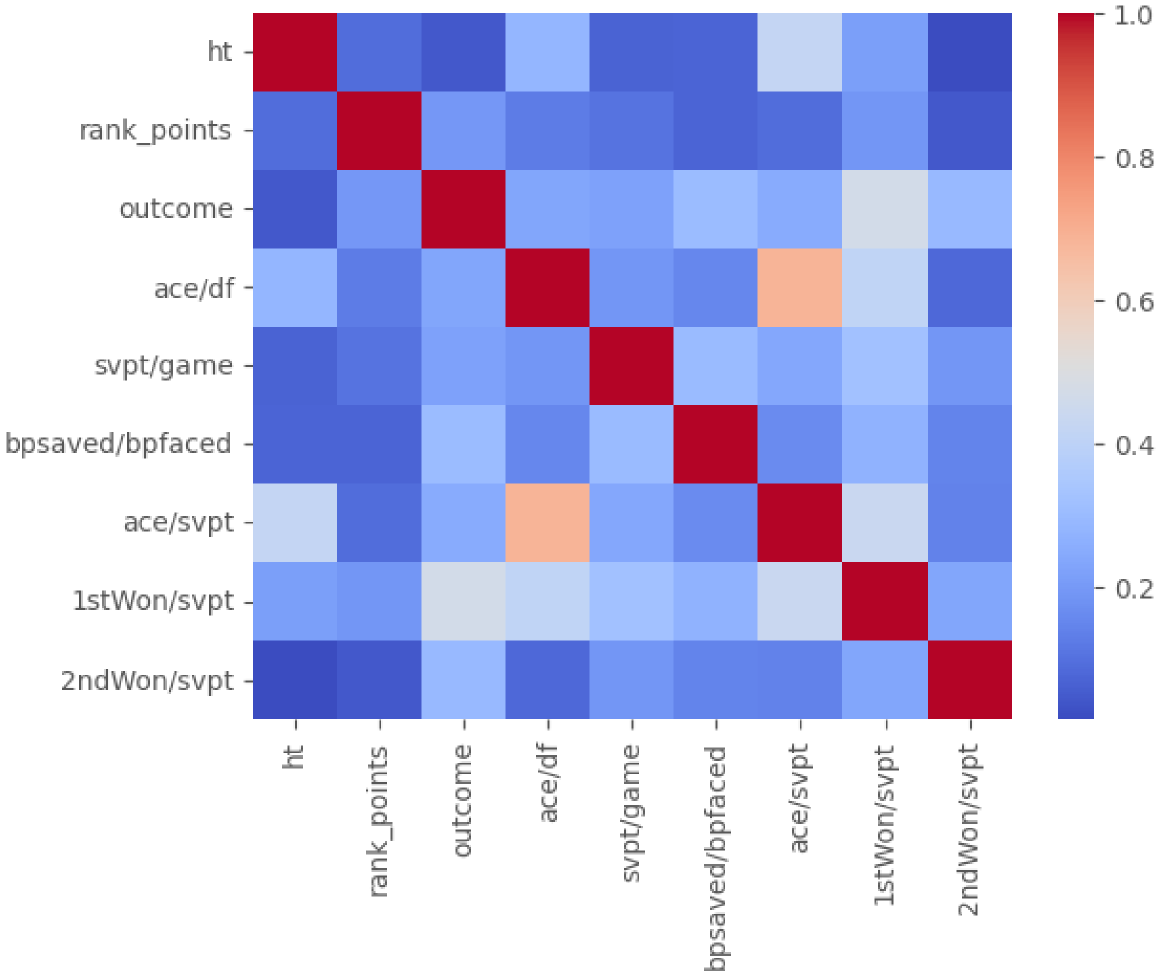
```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 125143 entries, 0 to 139129
Data columns (total 10 columns):
 #   Column           Non-Null Count   Dtype
---  ------           --------------   -----
 0   hand             125143 non-null  object
 1   ht               125143 non-null  float64
 2   rank_points      125143 non-null  float64
 3   outcome          125143 non-null  int64
 4   ace/df           123810 non-null  float64
 5   svpt/game        125136 non-null  float64
 6   bpsaved/bpfaced  118843 non-null  float64
 7   ace/svpt         125134 non-null  float64
 8   1stWon/svpt      125134 non-null  float64
 9   2ndWon/svpt      125134 non-null  float64
dtypes: float64(8), int64(1), object(1)
memory usage: 10.5+ MB
```

After the feature engineering is completed, we removed all of the raw statistical features, only keeping hand, height, and rank_points from the original features.

```
# Correlation matrix
corr_matrix = df_final.corr().abs()
sns.heatmap(corr_matrix, cmap='coolwarm')
print(corr_matrix['outcome'].sort_values(ascending=False))
```

```
outcome            1.000000
1stWon/svpt        0.466317
bpsaved/bpfaced    0.302502
2ndWon/svpt        0.292835
ace/svpt           0.248447
ace/df             0.231322
```

```
svpt/game          0.221659
rank_points        0.195144
ht                 0.042457
Name: outcome, dtype: float64
```



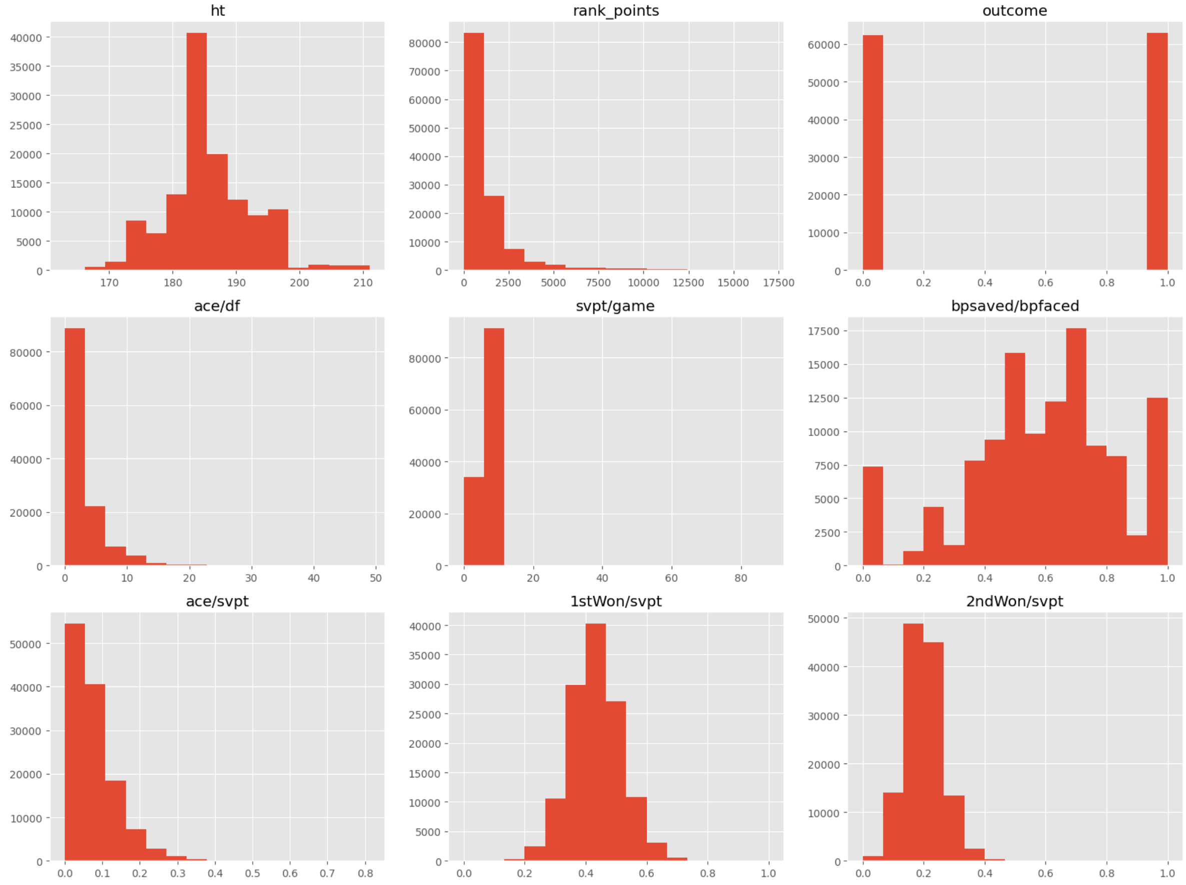
Summary Statistics

Below, we'll check out the summary statistics as well as histograms to paint the initial picture of our data and look for any glaring issues.

```
# Summary statistics on our features, making sure there isn't any obvious problems with th
df_final.describe().transpose()
```

	count	mean	std	min	25%	50%	75%	ma
ht	125143.0	185.906059	6.807160	163.0	183.000000	185.000000	190.000000	21
rank_points	125143.0	1332.367947	1668.267937	1.0	531.000000	835.000000	1400.000000	16
outcome	125143.0	0.502609	0.499995	0.0	0.000000	1.000000	1.000000	1.0
ace/df	123810.0	2.868646	3.216605	0.0	1.000000	2.000000	4.000000	49
svpt/game	125136.0	6.389549	0.897368	0.0	5.800000	6.315789	6.884615	88
bpsaved/bpfaced	118843.0	0.582380	0.251537	0.0	0.444444	0.600000	0.750000	1.0
ace/svpt	125134.0	0.076540	0.061411	0.0	0.032258	0.062500	0.106061	0.8
1stWon/svpt	125134.0	0.432407	0.084819	0.0	0.376344	0.430380	0.486486	1.0
2ndWon/svpt	125134.0	0.200130	0.060002	0.0	0.160000	0.196970	0.237288	1.0

```
# Histograms to check out distributions of different features, most of our crafted feature
df_final.hist(bins=15, figsize=(16, 12))
plt.tight_layout()
plt.show()
```



```
# Create x,y
X = df_final.drop('outcome', axis=1)
y = df_final['outcome']

print(f'X Shape: {X.shape}')
print(f'y Shape: {y.shape}')
```

```
X Shape: (125143, 9)
y Shape: (125143,)
```

Train/Test Split and Scaling

For the train/test split we chose a test size of 0.3, since our dataset is fairly large we feel confident training on 70% and testing on 30%.

```
#split X and y into train and test sets
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=4206
```

```
#standardize the train and test sets
from sklearn.pipeline import make_pipeline
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import make_column_selector, make_column_transformer

num_pipeline = make_pipeline(SimpleImputer(strategy="median"), StandardScaler())

cat_pipeline = make_pipeline(SimpleImputer(strategy="most_frequent"),OneHotEncoder(handle_

preprocessing = make_column_transformer(
    (num_pipeline, make_column_selector(dtype_include=np.number)),
    (cat_pipeline, make_column_selector(dtype_include=object)))
X_train_processed = preprocessing.fit_transform(X_train)
X_test_processed = preprocessing.transform(X_test)

print(f'X_train_processed shape: {X_train_processed.shape}')
print(f'X_test_processed shape: {X_test_processed.shape}')
```

```
X_train_processed shape: (87600, 11)
```

```
X_test_processed shape: (37543, 11)
```

Models

The code below defines functions to view model results, including the hyperparameters used and their accuracies as well as the confusion matrix and classification report.

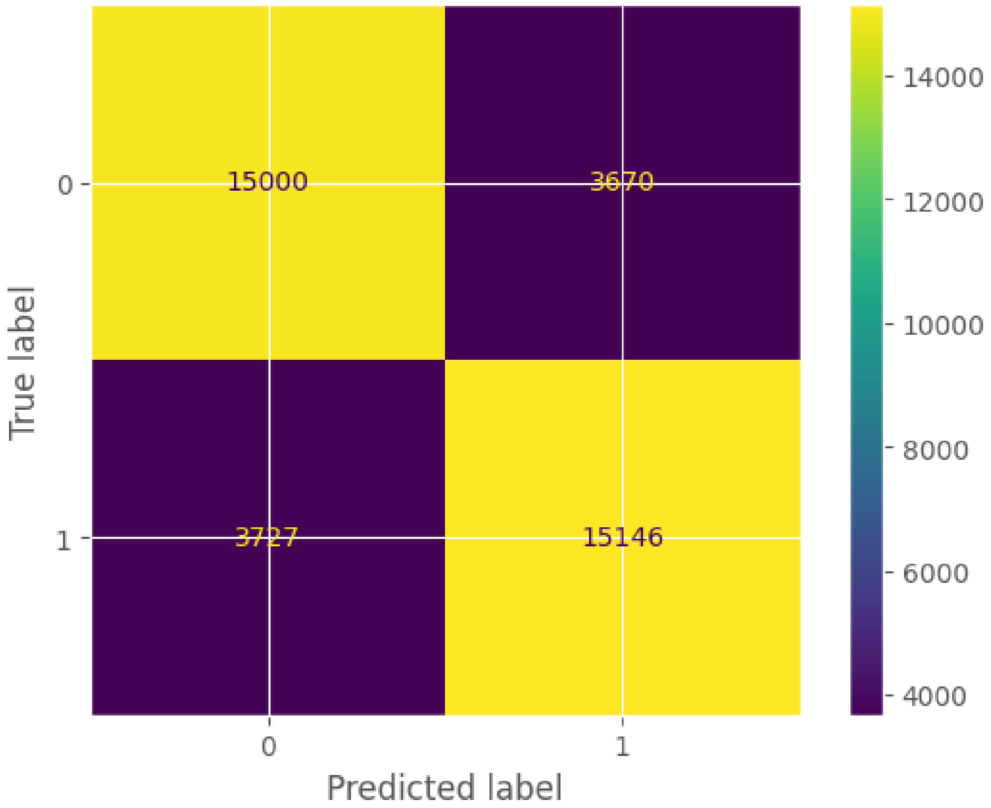
```
# Function to get results
from sklearn.metrics import classification_report,confusion_matrix, ConfusionMatrixDisplay

def ResultsOut(Model):
    y_pred = Model.predict(X_test_processed)
    disp =ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_test,y_pred))
    disp.plot()
    print(classification_report(y_test,y_pred))

def paramResults(mod):
    results_df = pd.DataFrame({'param': mod.cv_results_["params"], 'Accuracy': mod.cv_resu

    # Splitting the 'param' column into separate columns
    params_df = pd.DataFrame(results_df['param'].to_list())
    params_df.columns = [f"{col}_param" for col in params_df.columns]

    # Concatenating the new columns with the 'Accuracy' column
    results_df = pd.concat([params_df, results_df['Accuracy']], axis=1)

    # Displaying the updated DataFrame
    print(results_df)
```

Unregularized Logistic Regression

```
# Unregularized Logistic Regression
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

logreg = LogisticRegression(penalty=None)
logreg.fit(X_train_processed, y_train)

print(f'Unregularized Logistic Regression Accuracy: {logreg.score(X_test_processed, y_test
Unregularized Logistic Regression Accuracy: 0.8029725914284953
```

```
ResultsOut(logreg)
```

```
              precision    recall  f1-score   support

           0       0.80      0.80      0.80     18670
           1       0.80      0.80      0.80     18873

    accuracy                           0.80     37543
   macro avg       0.80      0.80      0.80     37543
weighted avg       0.80      0.80      0.80     37543
```



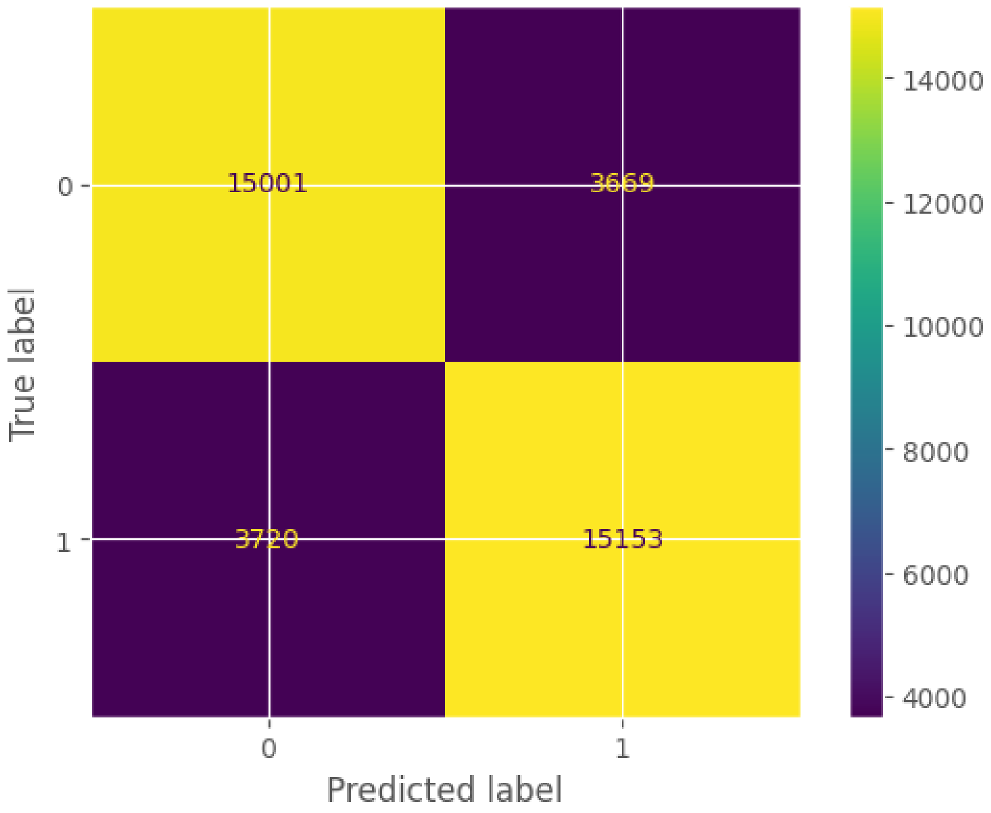
Ridge

```
# Ridge
# Need to find best params for param grid
start = time.perf_counter()
C_grid = [10**-3,10**-2,10**-1,10**0,10**1,10**2]

param_grid={'C':C_grid}
ridge_cv = GridSearchCV(LogisticRegression(penalty='l2',solver='saga'),
                                         param_grid=param_grid,cv=5)
ridge_cv.fit(X_train_processed,y_train)

print(ridge_cv.best_estimator_)
print(f'Ridge Accuracy: {ridge_cv.score(X_test_processed, y_test)}')
ridge_cv.cv_results_
paramResults(ridge_cv)
print(f'Runtime: {round(time.perf_counter()-start,3)}s')
```

```
LogisticRegression(C=1, solver='saga')
Ridge Accuracy: 0.8031856804197853
   C_param  Accuracy
0    0.001  0.797979
1    0.010  0.800947
2    0.100  0.801336
3    1.000  0.801381
4   10.000  0.801358
5  100.000  0.801358
Runtime: 19.569s
```

```
ResultsOut(ridge_cv)
```

```
              precision    recall  f1-score   support

           0       0.80      0.80      0.80     18670
           1       0.81      0.80      0.80     18873

    accuracy                           0.80     37543
   macro avg       0.80      0.80      0.80     37543
weighted avg       0.80      0.80      0.80     37543
```



Lasso

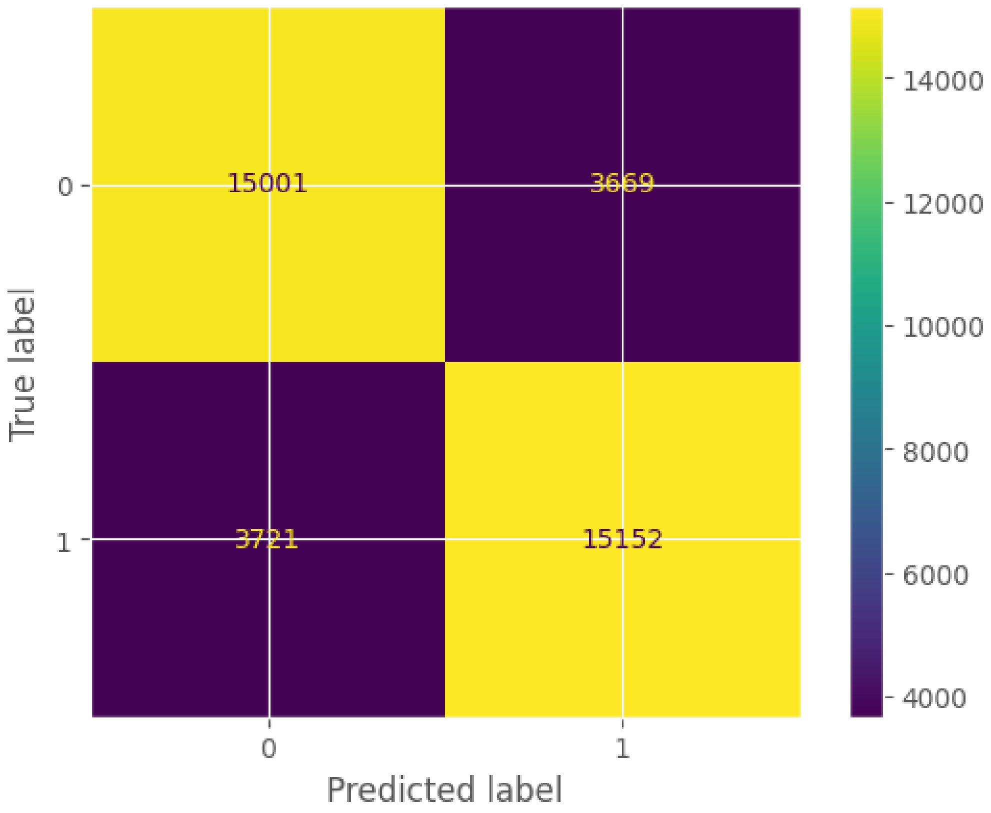
```
# Lasso
start = time.perf_counter()
param_grid={'C':C_grid}
lasso_cv = GridSearchCV(LogisticRegression(penalty='l1',solver='saga'),
                                          param_grid=param_grid,cv=5)
lasso_cv.fit(X_train_processed,y_train)

print(lasso_cv.best_estimator_)
print(f'Lasso Accuracy: {lasso_cv.score(X_test_processed, y_test)}')
lasso_cv.cv_results_
paramResults(lasso_cv)
print(f'Runtime: {round(time.perf_counter()-start,2)}')
```

```
LogisticRegression(C=10, penalty='l1', solver='saga')
Lasso Accuracy: 0.803159044295874
```

```
   C_param  Accuracy
0    0.001  0.797831
1    0.010  0.800845
2    0.100  0.801336
3    1.000  0.801358
4   10.000  0.801358
5  100.000  0.801358
Runtime: 22.17
```

```
ResultsOut(lasso_cv)
```

```
              precision    recall  f1-score   support

           0       0.80      0.80      0.80     18670
           1       0.81      0.80      0.80     18873

    accuracy                           0.80     37543
   macro avg       0.80      0.80      0.80     37543
weighted avg       0.80      0.80      0.80     37543
```



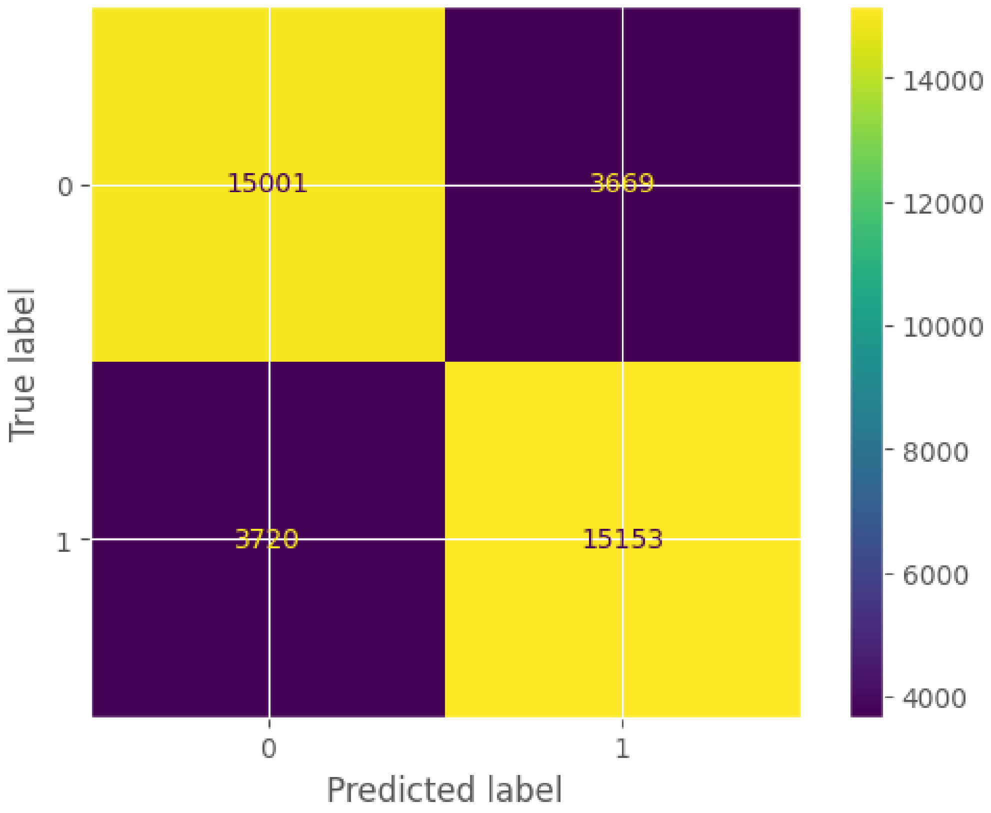
ElasticNet

```
# ElasticNet
start = time.perf_counter()

param_grid2={'C':C_grid,'l1_ratio':[.25, .5, .75]}

elastic_cv = GridSearchCV(LogisticRegression(penalty='elasticnet',solver='saga'),
                                            param_grid=param_grid2,cv=5)
elastic_cv.fit(X_train_processed,y_train)

print(elastic_cv.best_estimator_)
print(f'ElasticNet Accuracy: {elastic_cv.score(X_test_processed, y_test)}')
elastic_cv.cv_results_
paramResults(elastic_cv)
print(f'Runtime: {round(time.perf_counter()-start,3)}s')
```

```
LogisticRegression(C=1, l1_ratio=0.25, penalty='elasticnet', solver='saga')
ElasticNet Accuracy: 0.8031856804197853
    C_param  l1_ratio_param  Accuracy
0     0.001            0.25  0.798345
1     0.001            0.50  0.798550
2     0.001            0.75  0.798037
3     0.010            0.25  0.801016
4     0.010            0.50  0.800913
5     0.010            0.75  0.800845
6     0.100            0.25  0.801301
7     0.100            0.50  0.801279
8     0.100            0.75  0.801313
9     1.000            0.25  0.801370
10    1.000            0.50  0.801347
11    1.000            0.75  0.801347
12   10.000            0.25  0.801370
13   10.000            0.50  0.801358
14   10.000            0.75  0.801358
15  100.000            0.25  0.801358
16  100.000            0.50  0.801358
17  100.000            0.75  0.801358
Runtime: 66.438s
```

```
ResultsOut(elastic_cv)
```

```
              precision    recall  f1-score   support

           0       0.80      0.80      0.80     18670
           1       0.81      0.80      0.80     18873

    accuracy                           0.80     37543
   macro avg       0.80      0.80      0.80     37543
weighted avg       0.80      0.80      0.80     37543
```



K-Nearest Neighbours

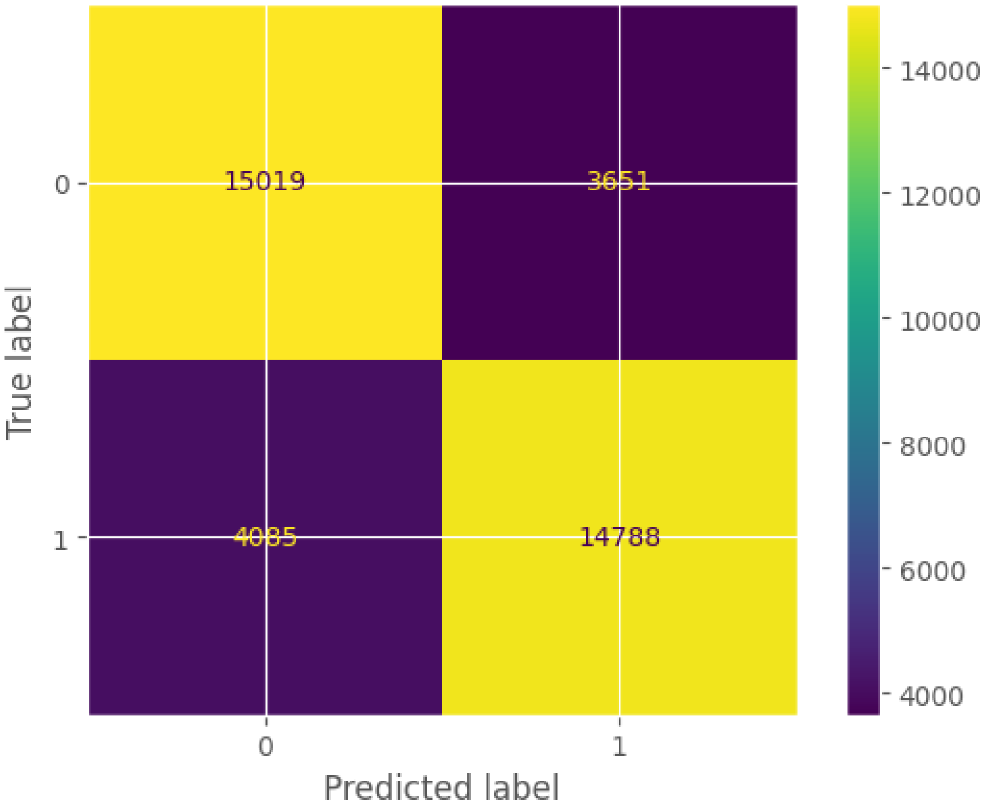
```
# K-Nearest Neighbours
from sklearn.neighbors import KNeighborsClassifier
start = time.perf_counter()

k_values = list(range(50,200,10))
param_grid = {'n_neighbors': k_values}

knn_cv = GridSearchCV(KNeighborsClassifier(),param_grid,cv=5,n_jobs=-1)
knn_cv.fit(X_train_processed,y_train)

print(knn_cv.best_estimator_)
print(f'K-Nearest Neighbors Accuracy: {knn_cv.score(X_test_processed, y_test)}')
knn_cv.cv_results_
paramResults(knn_cv)
```

```
print(f'Runtime: {round(time.perf_counter()-start,3)}s')
```

```
KNeighborsClassifier(n_neighbors=170)
K-Nearest Neighbors Accuracy: 0.7939429454225821
    n_neighbors_param  Accuracy
0                  50  0.792797
1                  60  0.792500
2                  70  0.792500
3                  80  0.793105
4                  90  0.793253
5                 100  0.792922
6                 110  0.792991
7                 120  0.793082
8                 130  0.793653
9                 140  0.793276
10                150  0.793447
11                160  0.793893
12                170  0.793950
13                180  0.793413
14                190  0.793505
Runtime: 150.23s
```

```
ResultsOut(knn_cv)
```

```
              precision    recall  f1-score   support

           0       0.79      0.80      0.80     18670
           1       0.80      0.78      0.79     18873

    accuracy                           0.79     37543
   macro avg       0.79      0.79      0.79     37543
weighted avg       0.79      0.79      0.79     37543
```



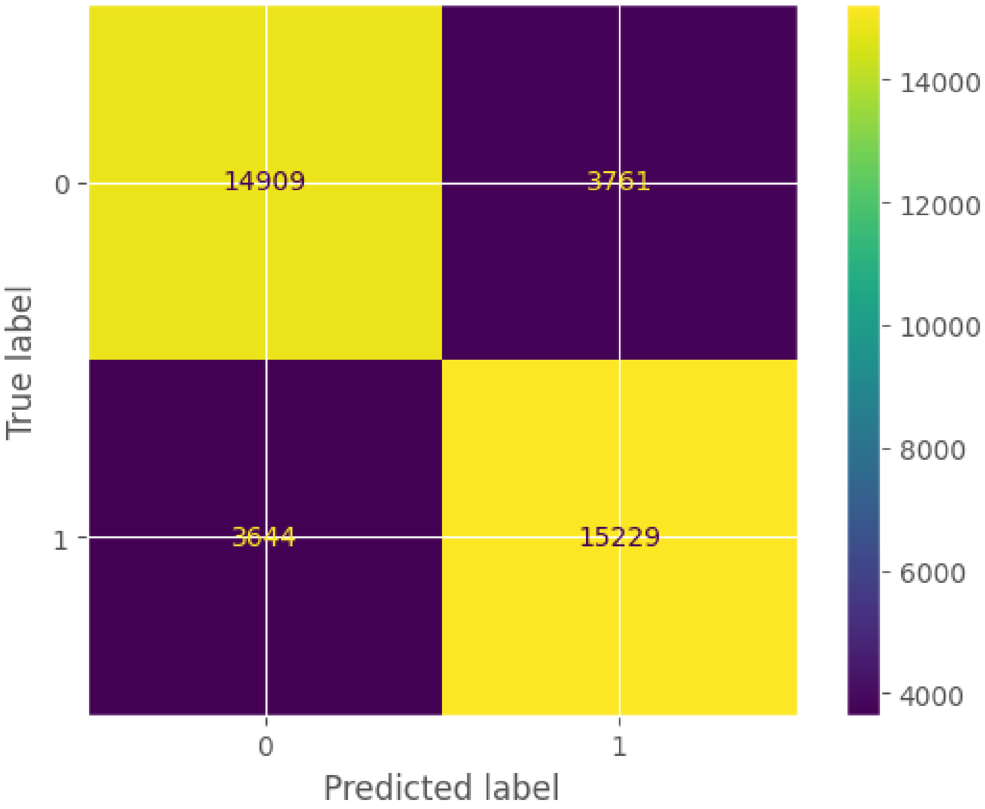
Random Forest

```
# Random Forest
from sklearn.ensemble import RandomForestClassifier
start = time.perf_counter()

param_grid_rf = {'n_estimators': [100,300,500],'max_depth': [10, 15, 20]}
rf_cv = GridSearchCV(estimator=RandomForestClassifier(random_state=42069,max_features='sqr
                                                     cv=5, param_grid=param_grid_rf,n_jobs
rf_cv.fit(X_train_processed, y_train)

print(rf_cv.best_estimator_)
print(f'Random Forest Accuracy: {rf_cv.score(X_test_processed, y_test)}')
rf_cv.cv_results_
paramResults(rf_cv)
print(f'Runtime: {round(time.perf_counter()-start,3)}s')
```

```
RandomForestClassifier(max_depth=15, n_estimators=500, random_state=42069)
Random Forest Accuracy: 0.8027595024372053
   max_depth_param  n_estimators_param  Accuracy
0               10                 100  0.799532
1               10                 300  0.799909
2               10                 500  0.799772
3               15                 100  0.800753
4               15                 300  0.801233
5               15                 500  0.801279
6               20                 100  0.798984
7               20                 300  0.800434
8               20                 500  0.800365
Runtime: 249.116s
```

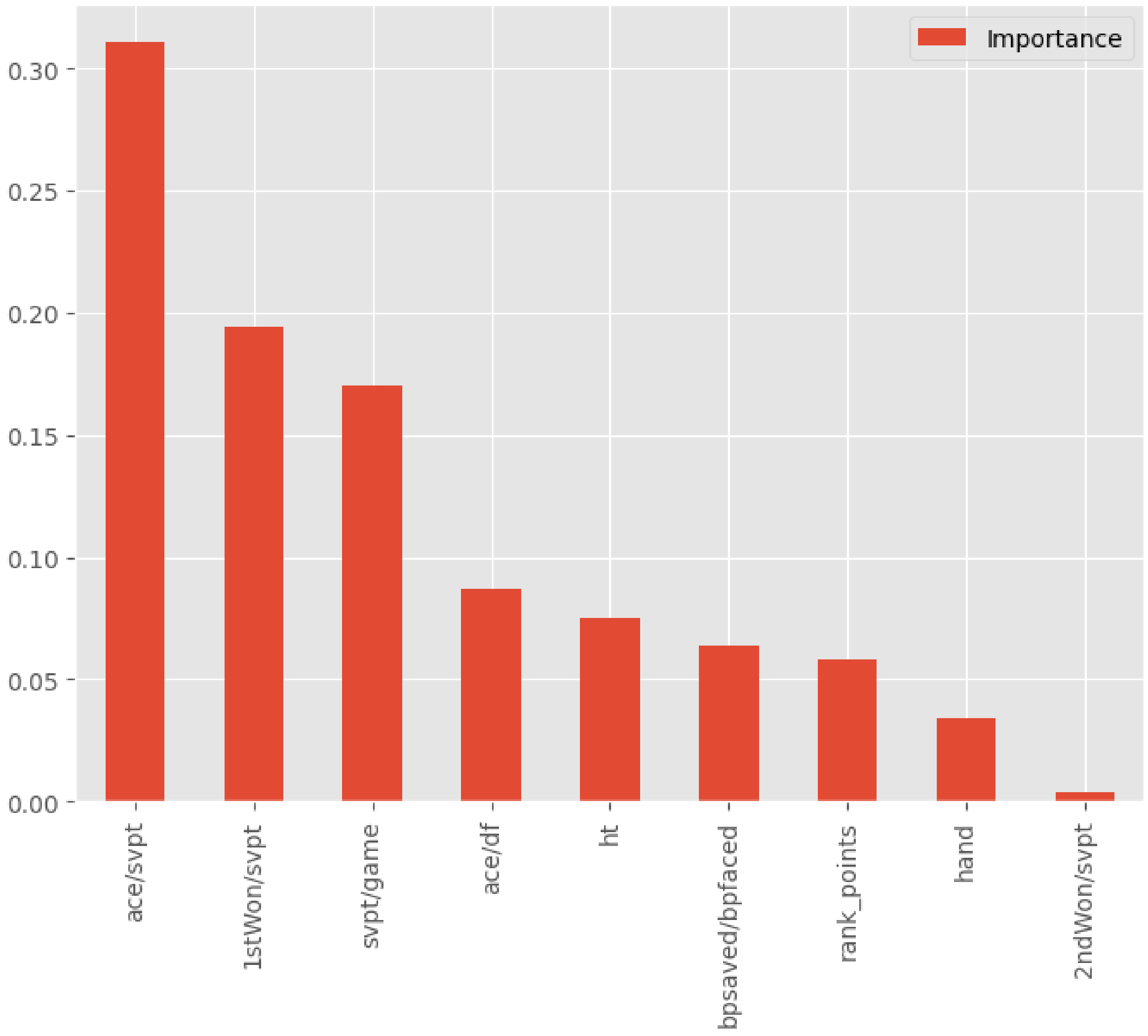
```
ResultsOut(rf_cv)
```

```
              precision    recall  f1-score   support

           0       0.80      0.80      0.80     18670
           1       0.80      0.81      0.80     18873

    accuracy                           0.80     37543
   macro avg       0.80      0.80      0.80     37543
weighted avg       0.80      0.80      0.80     37543
```



```
feat_importances = pd.DataFrame(rf_cv.best_estimator_.feature_importances_[:9], index=X_tr
feat_importances.sort_values(by='Importance', ascending=False, inplace=True)
feat_importances.plot(kind='bar', figsize=(8,6))
```

```
<AxesSubplot: >
```



Gradient Boosted Trees

```
# Gradient Boosted Trees
from sklearn.ensemble import GradientBoostingClassifier

start = time.perf_counter()
param_grid_gbrt = {'n_estimators': [100,300,500],'learning_rate': [0.01,0.1],'max_depth':
GBRT = GridSearchCV(estimator=GradientBoostingClassifier(random_state=42069),
                    cv=5, param_grid=param_grid_gbrt,n_jobs=-1)
GBRT.fit(X_train_processed, y_train)
```

```
print(GBRT.best_estimator_)
print(f'Gradient Boosed Trees Accuracy: {GBRT.score(X_test_processed, y_test)}')
GBRT.cv_results_
paramResults(GBRT)
print(f'Runtime: {round(time.perf_counter()-start,3)}s')
```

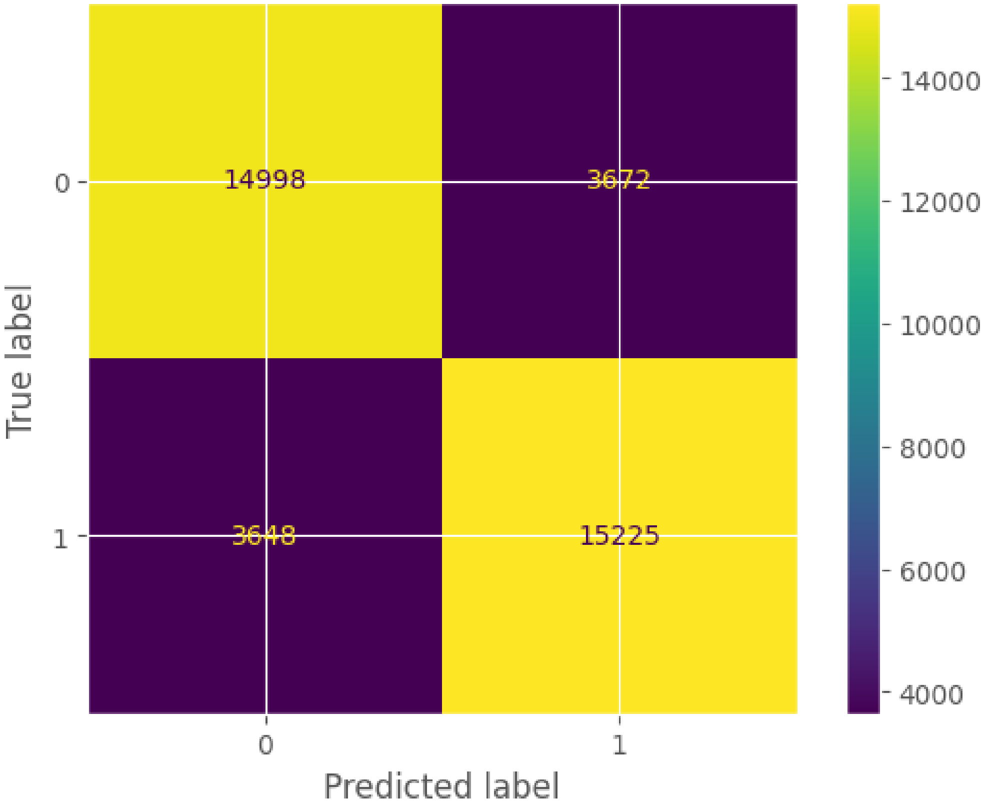
```
GradientBoostingClassifier(max_depth=2, n_estimators=500, random_state=42069,
                           subsample=0.5)
Gradient Boosed Trees Accuracy: 0.8050235729696614
    learning_rate_param  max_depth_param  n_estimators_param  subsample_param  \
0                  0.01                2                 100              0.5
1                  0.01                2                 100              1.0
2                  0.01                2                 300              0.5
3                  0.01                2                 300              1.0
4                  0.01                2                 500              0.5
5                  0.01                2                 500              1.0
6                  0.01                4                 100              0.5
7                  0.01                4                 100              1.0
8                  0.01                4                 300              0.5
9                  0.01                4                 300              1.0
10                 0.01                4                 500              0.5
11                 0.01                4                 500              1.0
12                 0.01                6                 100              0.5
13                 0.01                6                 100              1.0
14                 0.01                6                 300              0.5
15                 0.01                6                 300              1.0
16                 0.01                6                 500              0.5
17                 0.01                6                 500              1.0
18                 0.10                2                 100              0.5
19                 0.10                2                 100              1.0
20                 0.10                2                 300              0.5
21                 0.10                2                 300              1.0
22                 0.10                2                 500              0.5
23                 0.10                2                 500              1.0
24                 0.10                4                 100              0.5
25                 0.10                4                 100              1.0
26                 0.10                4                 300              0.5
27                 0.10                4                 300              1.0
28                 0.10                4                 500              0.5
29                 0.10                4                 500              1.0
30                 0.10                6                 100              0.5
```

```
31                  0.10                6                  100                1.0
32                  0.10                6                  300                0.5
33                  0.10                6                  300                1.0
34                  0.10                6                  500                0.5
35                  0.10                6                  500                1.0

    Accuracy
0   0.778185
1   0.776975
2   0.791826
3   0.791575
4   0.796336
5   0.796005
6   0.791644
7   0.791393
8   0.798550
9   0.798676
10  0.800548
11  0.800765
12  0.797374
13  0.796438
14  0.802260
15  0.801062
16  0.803185
17  0.802568
18  0.801119
19  0.800342
20  0.802877
21  0.802877
22  0.803322
23  0.803322
24  0.802877
25  0.802546
26  0.802409
27  0.802842
28  0.801301
29  0.801655
30  0.801678
31  0.802900
32  0.797329
33  0.801393
34  0.794749
35  0.799178
```

```
Runtime: 994.774s
```

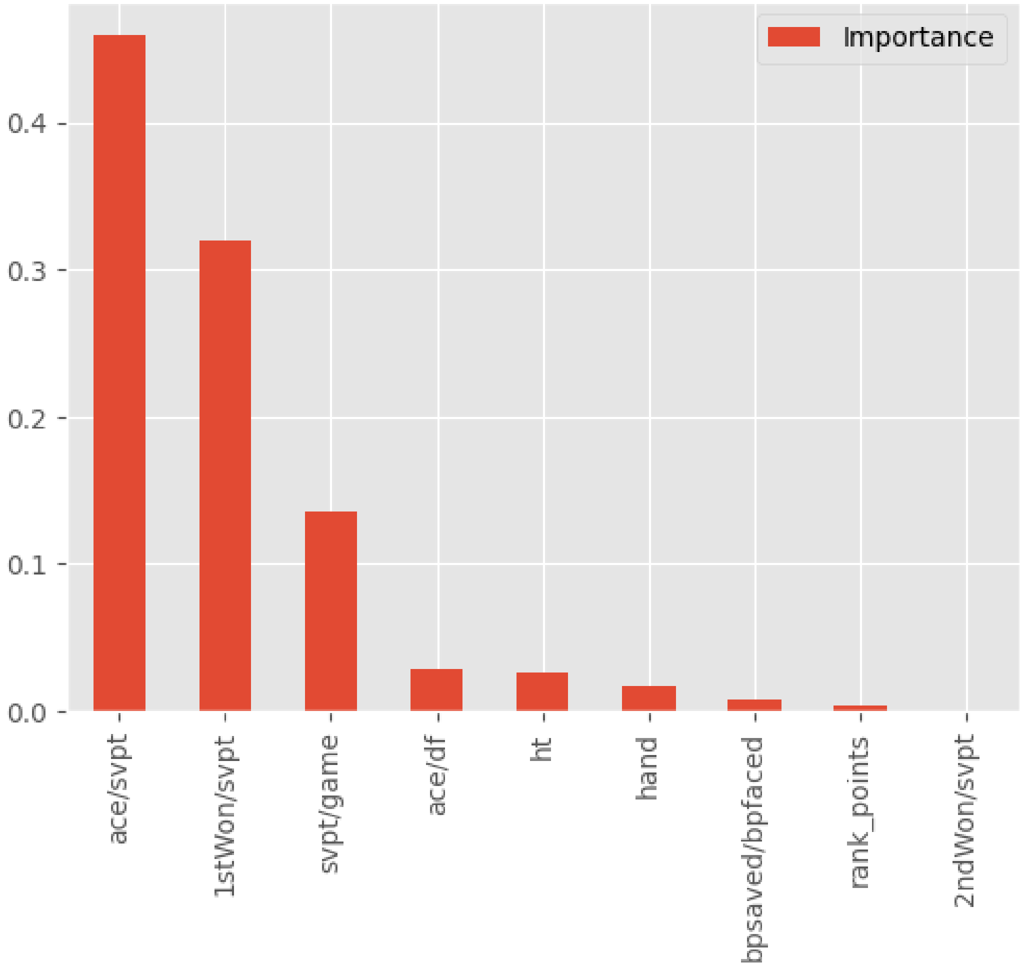
```
ResultsOut(GBRT)
```

```
              precision    recall  f1-score   support

           0       0.80      0.80      0.80     18670
           1       0.81      0.81      0.81     18873

    accuracy                           0.81     37543
   macro avg       0.81      0.81      0.81     37543
weighted avg       0.81      0.81      0.81     37543
```



```
feat_importances = pd.DataFrame(GBRT.best_estimator_.feature_importances_[:9], index=X_tra
feat_importances.sort_values(by='Importance', ascending=False, inplace=True)
feat_importances.plot(kind='bar')
```

```
<AxesSubplot: >
```



XGBoost

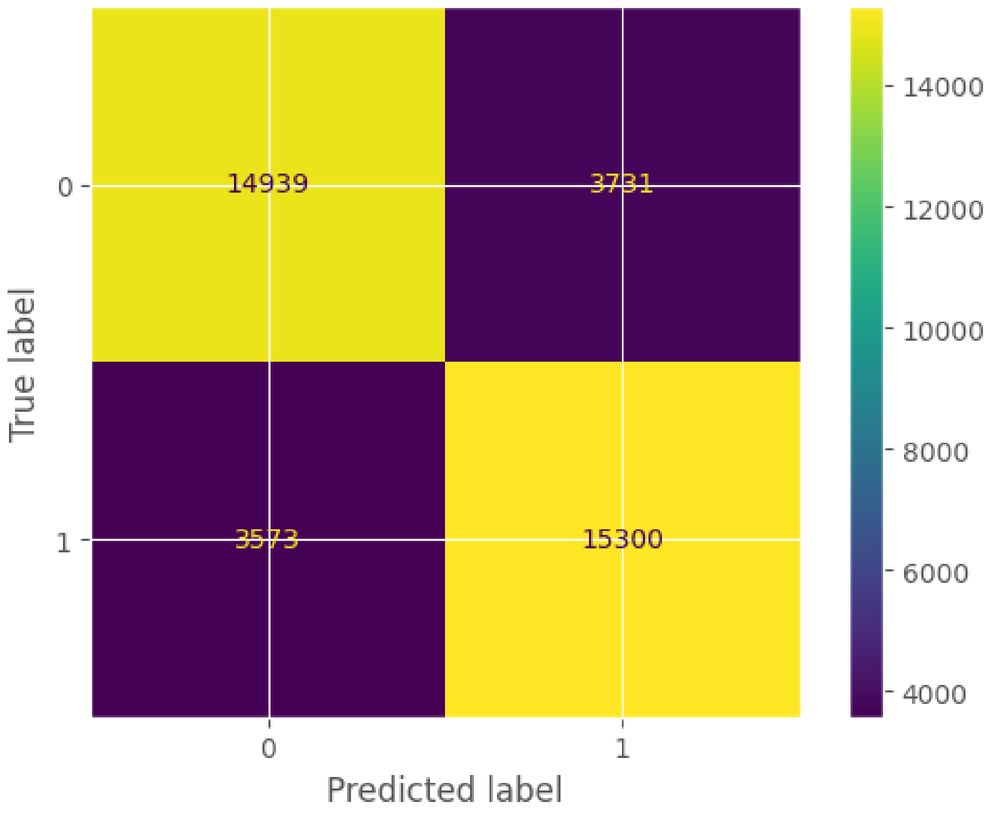
```
from xgboost import XGBClassifier
start = time.perf_counter()

param_grid_xgb = {'n_estimators': [300,500,700],'learning_rate': [0.01,0.1],'max_depth': [
XGB = GridSearchCV(estimator=XGBClassifier(random_state=42069), cv=5, param_grid=param_gri
XGB.fit(X_train_processed, y_train)

print(XGB.best_estimator_)
print(f'XGBoost Accuracy: {XGB.score(X_test_processed, y_test)}')
paramResults(XGB)
print(f'Runtime: {round(time.perf_counter()-start,3)}s')
```

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric=None, feature_types=None,
              gamma=None, gpu_id=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=0.1, max_bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=3, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constraints=None,
              n_estimators=300, n_jobs=None, num_parallel_tree=None,
              predictor=None, random_state=42069, ...)
XGBoost Accuracy: 0.8054497509522415
    learning_rate_param  max_depth_param  n_estimators_param  Accuracy
0                  0.01                3                 300  0.794737
1                  0.01                3                 500  0.797968
2                  0.01                3                 700  0.800057
3                  0.01                5                 300  0.798527
4                  0.01                5                 500  0.802043
5                  0.01                5                 700  0.802957
6                  0.01                7                 300  0.801416
7                  0.01                7                 500  0.802283
8                  0.01                7                 700  0.802180
9                  0.10                3                 300  0.803082
10                 0.10                3                 500  0.802717
11                 0.10                3                 700  0.802648
12                 0.10                5                 300  0.802489
13                 0.10                5                 500  0.801872
14                 0.10                5                 700  0.800342
15                 0.10                7                 300  0.800890
16                 0.10                7                 500  0.799486
17                 0.10                7                 700  0.797671
Runtime: 406.398s
```

```
ResultsOut(XGB)
```

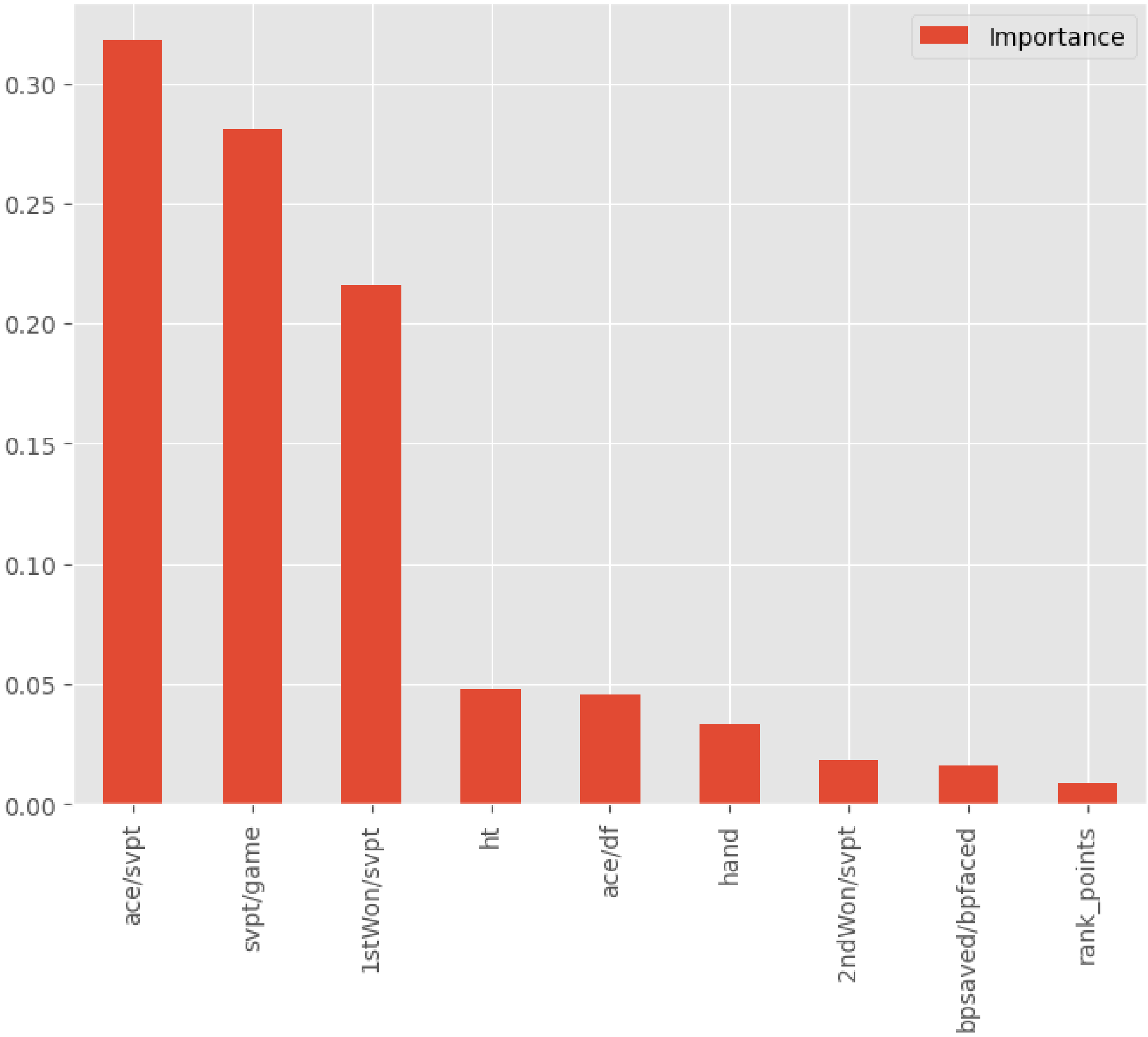
```
              precision    recall  f1-score   support
```

```
           0       0.81      0.80      0.80     18670
           1       0.80      0.81      0.81     18873

    accuracy                           0.81     37543
   macro avg       0.81      0.81      0.81     37543
weighted avg       0.81      0.81      0.81     37543
```



```
feat_importances = pd.DataFrame(XGB.best_estimator_.feature_importances_[:9], index=X_trai
feat_importances.sort_values(by='Importance', ascending=False, inplace=True)
feat_importances.plot(kind='bar', figsize=(8,6))
```

```
<AxesSubplot: >
```


Importance
0.30
0.25
0.20
0.15
0.10
0.05
0.00
ace/svpt
svpt/game
1stWon/svpt
ht
ace/df
hand
2ndWon/svpt
bpsaved/bpfaced
rank_points