ISYE 7406: HW 5

Matt McDowell

March 2025

1 Abstract

This paper assess the effectiveness of two types different ensemble methods: ***random forests*** and ***boosting*** to create a classifier model to predict the results of professional tennis matches. These methods were compared against baseline models such as logistic regression and K-nearest-neighbors. Cross validation was used in order to tune the parameters of these models as well as evaluate the testing error and variance of their predictions. The findings demonstrate that the XGBoost model produced the best classifier and outperformed the other models. The results highlight the importance of using robust cross validation techniques to assess model performance and tune model parameters, additionally they highlight the learning capability of ensemble methods.

2 Introduction

Every year there are thousands of professional tennis matches played on almost every continent in the the world, generating large amounts of data on player performance. Famously, IBM was an early pioneer in using machine learning techniques as well as leveraging Watson to generate insights and make match predictions. However, compared to the early years there are now more advanced methods available such as *random forests* and *boosting* which leverage the combining many "weak" learners to create a single strong classifier. This paper explores the effectiveness of these models more traditional against baseline models to predict tennis match outcomes.

For those interested, but not familiar with tennis here is a general guide to the rules : Tennis Basics.

3 Problem Statement/Exploratory Analysis

3.1 Problem Statement

The dataset used in this analysis has match ATP (men's professional tennis) data from 1968 which is considered the beginning of the "Open Era" in men's professional tennis [1].

However, this dataset only has complete in-game match statistics after 1991 therefore data from after 1991 was exclusively used. The goal of this analysis is develop a prediction/classification model that can accurately determine wether a player will win or lose a match based on their performance. This will allow coaches and players to make strategic/tactical decisions to optimize their chances at winning the match.

3.2 Exploratory Analysis

In total, this dataset contains 26 features from over 183,000 professional matches,. Of these 26 features only 8 pertained to in-game match performance,these along with player height, player age, and court surface were used to train the classifiers. Additionally, I created a binary variable (win) that indicates wether the player won or lost the match. In this analysis the win variable will serve as the response and the rest of the data will be used as predictors.

For this analysis I selected 10,000 samples at random and created the following histograms (Figure 1) to show the distributions of the features. A breakdown of the meaning of each of these features is as follows:

1. ht - player height (cm)
2. age - player age (yrs)
3. surface - surface type (hard court, clay, grass, carpet)
4. ace - aces hit
5. df - double faults
6. svpt - overall serve percent
7. 1stIn - 1st serve in percent
8. 1stWon - 1st serve win percent
9. 2ndWon - 2nd serve win percent
10. bpSaved - break points saved
11. bpFaced - break points faced
12. win - result of match (1-win, 0-lose)

Additionally to determine the predictive potential of this dataset I created a correlation matrix (Figure 2). Based on the correlation matrix, there are many features with "weak" correlations (< 0.5) to but there isn't a single strong predictor. Therefore this dataset appears to be a good candidate for ensemble models specifically ***boosting*** where multiple weak predictors are combined to make a single strong classifier.

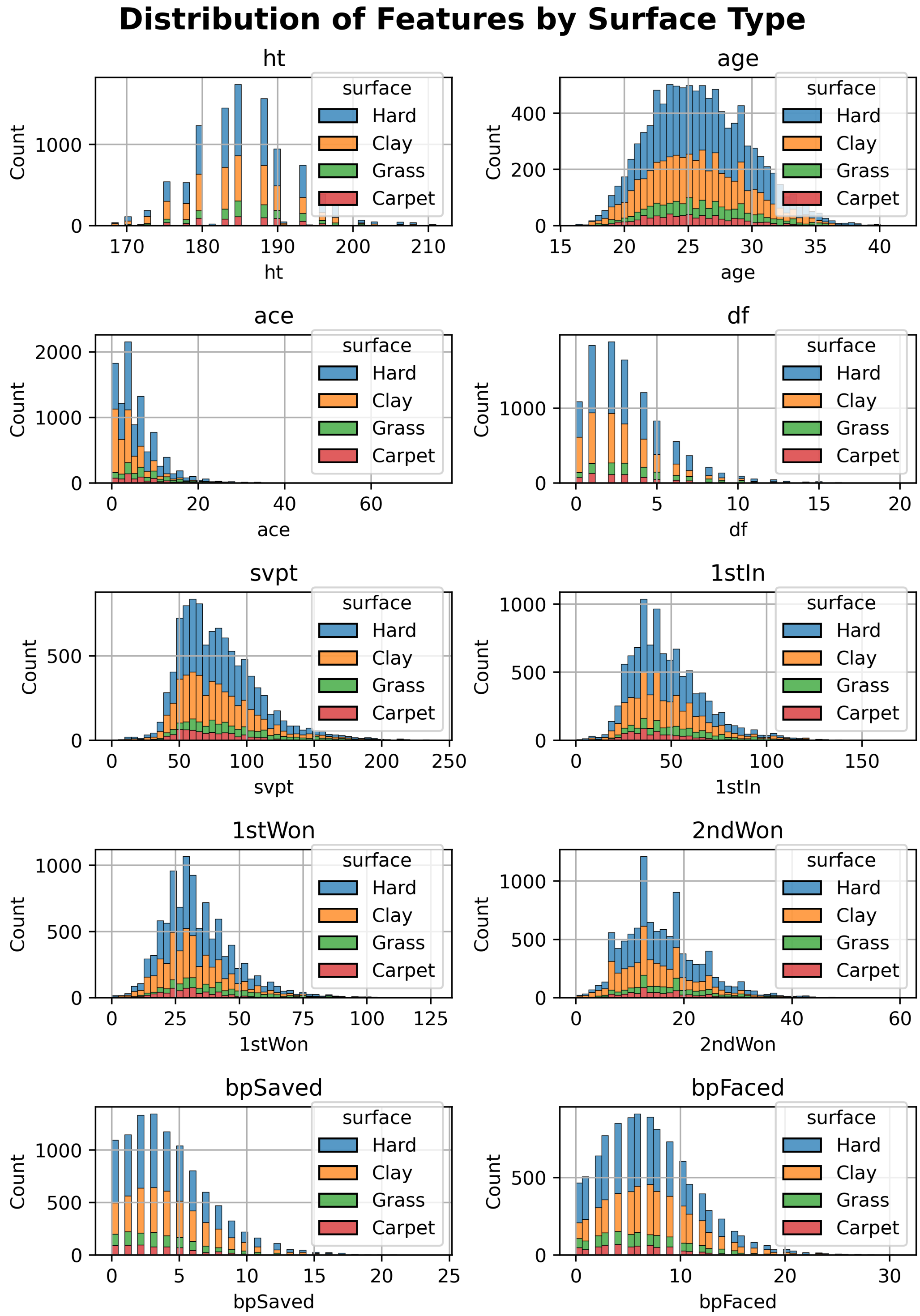

Figure 1: Distribution of Features by Surface Type

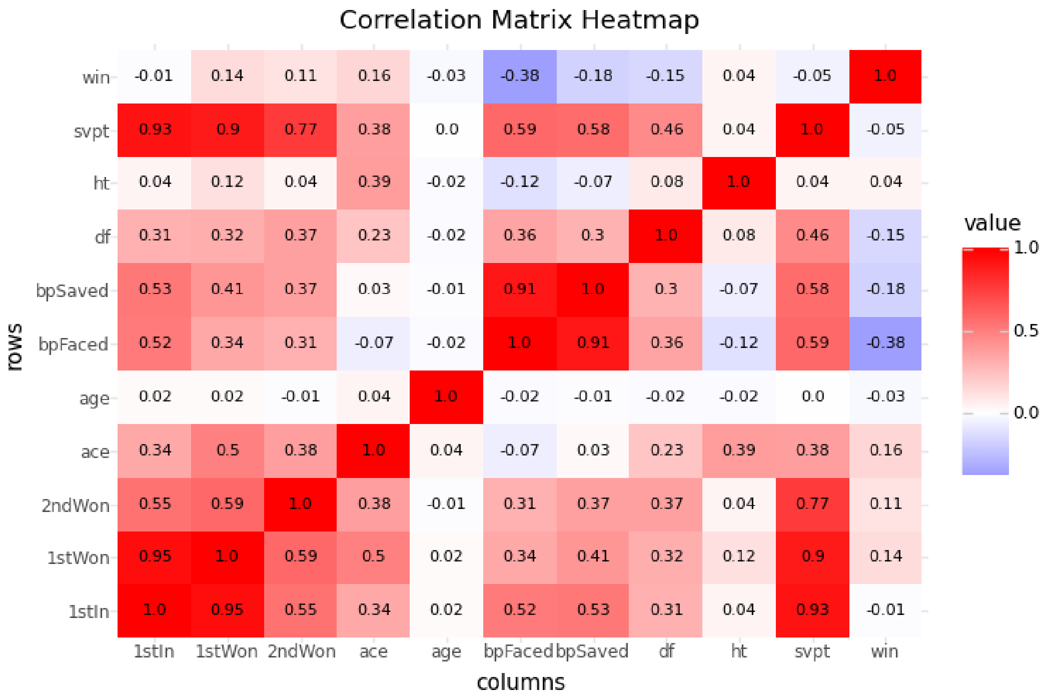

Figure 2: Correlation Matrix Heatmap

4 Methodology

In this section I will describe the different methods used to create/generate the ensemble classification models as well the baseline classification models. Additionally, I will briefly summarize their initial performance on the testing data and where applicable I will optimize model parameters using the testing error. Where applicable, when a decision boundary for predicting a class was needed the threshold value of 0.5 was chosen to give an equal weight to misclassifications of both wins and losses.

For this section the data was split (90/10) for training/testing and the same training and testing data was used for each model. When model parameters where optimized k-fold (k=10) cross validation was used.

4.1 Baseline Models

The baseline models that were chosen to test against the ensemble models are *Logistic Regression* and *K-Nearest-Neighbors*. These models were chosen because of their simplicity, interpretability, and widespread use in classification problems.

4.1.1 Logistic Regression

Logistic Regression is a linear model that estimates the probability of a given class using the logistic function.

Logit Regression Results

Dep. Variable:	win	No. Observations:	9000
Model:	Logit	Df Residuals:	8988
Method:	MLE	Df Model:	11
Date:	Thu, 13 Mar 2025	Pseudo R-squ.:	0.3456
Time:	17:43:24	Log-Likelihood:	-4082.2
converged:	True	LL-Null:	-6238.2
Covariance Type:	nonrobust	LLR p-value:	0.000

	coef	std err	z	P>|z|	[0.025	0.975]
const	-0.0717	0.028	-2.580	0.010	-0.126	-0.017
ht	-0.2082	0.030	-6.854	0.000	-0.268	-0.149
age	-0.1853	0.028	-6.661	0.000	-0.240	-0.131
ace	-0.3076	0.039	-7.817	0.000	-0.385	-0.230
df	-0.0243	0.036	-0.676	0.499	-0.095	0.046
svpt	-3.2817	0.268	-12.260	0.000	-3.806	-2.757
1stIn	-0.1982	0.166	-1.193	0.233	-0.524	0.127
1stWon	2.9752	0.181	16.462	0.000	2.621	3.329
2ndWon	1.4224	0.096	14.856	0.000	1.235	1.610
bpSaved	1.7917	0.103	17.404	0.000	1.590	1.994
bpFaced	-2.2650	0.145	-15.580	0.000	-2.550	-1.980
surface_num	-0.1126	0.028	-4.049	0.000	-0.167	-0.058

Figure 3: Logistic Regression Model

$$\log\left(\frac{P}{1-P}\right) = B_0 + B_1 x_{i,1} \ldots B_{p-1} x_{i,p-1} \tag{1}$$

The basic assumption of the Logistic Regression classification model is that it models conditional probability of the response directly and makes no assumptions about the distribution of predictors. For this baseline model all predictor variables were used, the intial training error and testing errors of this model were *0.201* and *0.206* respectively. See Figure 3 for a summary of the model.

4.1.2 K-Nearest-Neighbors

K-Nearest-Neighbors (KNN) is a classification model that uses distance metrics to assign data to a class based on its "k" nearest neighbors. It is another commonly used model since it is another non-parametric model, meaning that the model isn't making assumptions about the underlying data distributions. Since there was a relatively large amount of data I decided to use cross validation to optimize the k parameter by testing k-values from 1 to 100. After tuning this parameter the optimal model was KNN = 61, see Figure 4 . This model's training and testing errors were *0.259* and *0.277* respectively.

4.2 Ensemble Models

The ensemble models that were chosen for this analysis are random forests and XGBoost. Both of these models improve classification performance by by combining multiple weak learners. However, these models differ in their learning strategies: Random Forest uses on bagging, while XGBoost uses boosting.

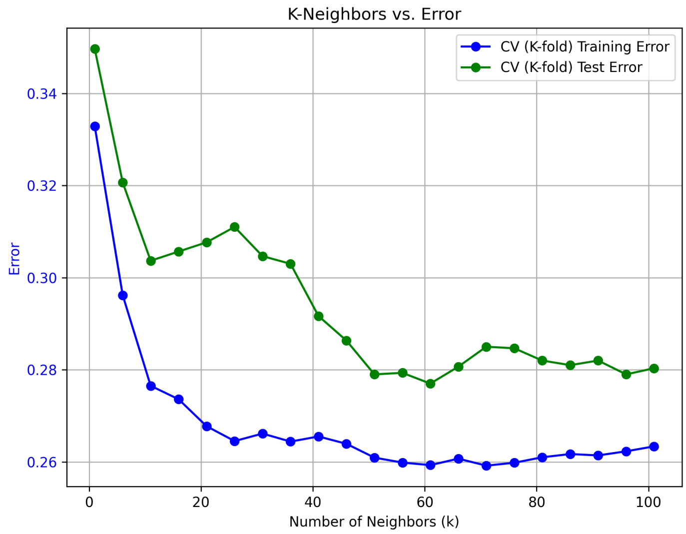

Figure 4: KNN Model

4.2.1 Random Forest

Random Forest is an ensemble learning method that uses decision trees with a random subset of features and bagging to decrease the variance in classification. The key idea in bagging is to use bootstrapped (randomly sampled w/replacement) data to make each tree and use the majority vote for classification. The random subset of features introduces diversity among the trees, leading to more robust and generalizable predictions. For this analysis I used cross validation to optimize the number of estimators, see Figure 5 . The optimal value for number number of estimators was found to be 81 and this model's training and testing errors were *0.292* and *0.281* respectively.

4.2.2 Boosting: XGBoost

XGBoost (Extreme Gradient Boosting) is a boosting ensemble method that builds decision trees sequentially, with each tree built correcting misclassifications from the previous tree. Unlike Random Forest, which grows trees independently XGBoost assigns greater weight to "weak" learners and the weighted average of the trees is used to improve the training error. For this analysis I used the number of estimators found for the random forest and used cross validation to optimize the learning rate, see Figure 6. The optimal value for the learning rate was found to be 0.21 and this model's training and testing errors were *0.212* and *0.248* respectively.

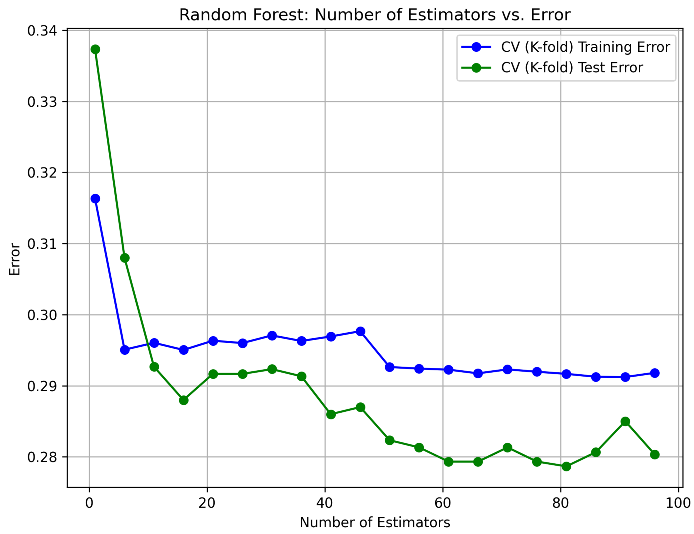

Figure 5: Random Forest Model

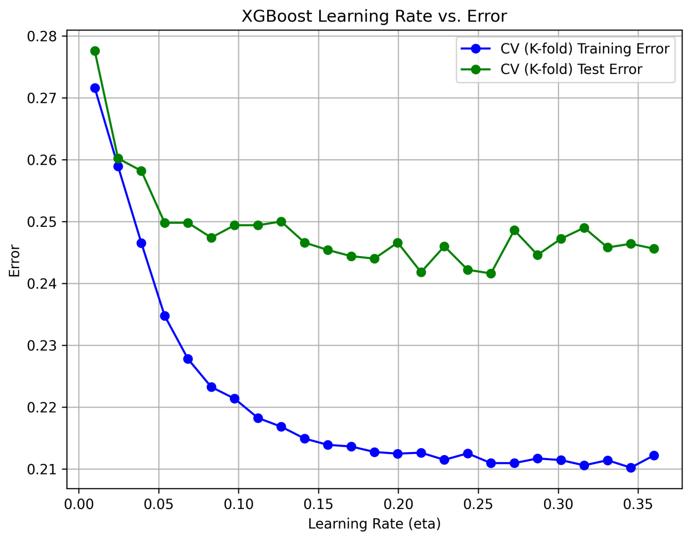

Figure 6: XGBoost Model

Model	Mean Test Error	Mean Test Variance
Logistic Regression	0.208	1.79E-4
KNN	0.250	1.12E-4
Random Forest	0.287	1.69E-4
XGBoost	0.187	1.60E-4

Table 1: Classification Model Comparison

5 Results/Analysis

This section contains a brief comparative analysis of the different models and their performance on dataset.

Initially, the classification models were evaluated on one split of of the data while the model parameters were selected using cross validation. In order to provide a more thorough comparison 100 Monte Carlo cross validation simulations were used to evaluate the testing errors and variances of each model, see Table 1. One of the most significant findings of this analysis is how much XGBoost improved from it original performance on the testing data set. It improved by from its performance on one split of the data by 22% to become the best model. This can be attributed to the fact that allowing XGBoost to see many partitions of the data it was better able to improve upon its weak learners effectively. In contrast, **Random Forest** showed much weaker performance, and its its improvement was much more modest compared to XGBoost.

6 Conclusions

In summary, out of the different models the XGBoost classification model was able to significantly outperform Random Forests as well as baseline models such as Logistic Regression and KNN. The improvement of the XGBoost model after 100 Monte Carlo simulations highlights the strengths of boosting techniques to improve the classification model.

However, while the final model is strong, it is still wrong approximately 20% of the time. This shows the inherent challenges with working with the complexity of professional tennis match data and fully capturing the underlying patterns.

6.1 Lessons Learned

An important lesson learned is that while boosting techniques like XGBoost can significantly enhance predictive performance, there is still room for improvement. Future work could be done to further fine-tune parameters or experimenting with alternative ensemble methods to reduce the error rate. Additionally, considering the cost of misclassifications in real-world applications, techniques such as adjusting the decision threshold or using cost-sensitive learning could be valuable in refining model performance. Finally, these results show the importance of using cross-validation techniques for both model selection and performance evaluation.

7 References

[1] Sijo Manikandan. *ATP matches*. URL: https://www.kaggle.com/datasets/sijovm/atpdata/data.

A Appendix A: Python Code

```
HW5 Ensemble Methods
--------------------
.. code:: ipython3

    import pandas as pd
    from datetime import datetime, timedelta
    import numpy as np
    import matplotlib.pyplot as plt
    import seaborn as sns
    from sklearn.preprocessing import LabelEncoder,StandardScaler,
MinMaxScaler, RobustScaler, Normalizer
    from sklearn import model_selection
    from sklearn.model_selection import train_test_split,RepeatedKFold
,cross_val_score
    from plotnine import ggplot, aes, geom_tile, scale_fill_gradient,
theme_minimal, theme, geom_boxplot
    from plotnine import ggsave, scale_y_reverse, labs,
scale_fill_gradient2,geom_line,geom_point,scale_color_manual,
geom_text
    import statsmodels.api as sm
    from sklearn.metrics import confusion_matrix,
ConfusionMatrixDisplay, accuracy_score, precision_score,roc_curve,
 auc
    from sklearn.neighbors import NearestNeighbors,
KNeighborsClassifier
    from sklearn.ensemble import GradientBoostingClassifier,
RandomForestClassifier
    import xgboost as xgb
    from tqdm import tqdm

.. code:: ipython3

    data = pd.read_csv("atp_matches_till_2022.csv")

.. code:: ipython3

    data['tourney_yr']  = data['tourney_id'].apply(lambda x: x.split('
-')[0]).astype(int)
    data = data.loc[data['tourney_yr'] >= 1991]

.. code:: ipython3

    data_win = data[['winner_id','winner_name','surface','winner_ht','
winner_age','w_ace', 'w_df','w_svpt','w_1stIn','w_1stWon','
w_2ndWon','w_bpSaved','w_bpFaced']].dropna()
```

```
    data_lose = data[['loser_id','loser_name','surface','loser_ht','loser_age','l_ace', 'l_df','l_svpt','l_1stIn','l_1stWon','l_2ndWon','l_bpSaved','l_bpFaced']].dropna()
    data_win.columns = [col.split('_', 1)[1] if '_' in col else col for col in data_win.columns]
    data_lose.columns = [col.split('_', 1)[1] if '_' in col else col for col in data_lose.columns]
    data_win['win'] = 1
    data_lose['win'] = 0

.. code:: ipython3

    dataset = pd.concat([data_win, data_lose], axis=0)
    dataset = dataset.sample(n=10000, random_state=42).reset_index(drop=True)

.. code:: ipython3

    dataset.describe()




.. raw:: html

    <div>
    <style scoped>
        .dataframe tbody tr th:only-of-type {
            vertical-align: middle;
        }

        .dataframe tbody tr th {
            vertical-align: top;
        }

        .dataframe thead th {
            text-align: right;
        }
    </style>
    <table border="1" class="dataframe">
      <thead>
        <tr style="text-align: right;">
          <th></th>
          <th>id</th>
          <th>ht</th>
          <th>age</th>
          <th>ace</th>
          <th>df</th>
          <th>svpt</th>
```

```
          <th>1stIn</th>
          <th>1stWon</th>
          <th>2ndWon</th>
          <th>bpSaved</th>
          <th>bpFaced</th>
          <th>win</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <th>count</th>
          <td>10000.000000</td>
          <td>10000.000000</td>
          <td>10000.000000</td>
          <td>10000.000000</td>
          <td>10000.000000</td>
          <td>10000.000000</td>
          <td>10000.000000</td>
          <td>10000.000000</td>
          <td>10000.000000</td>
          <td>10000.000000</td>
          <td>10000.000000</td>
          <td>10000.000000</td>
        </tr>
        <tr>
          <th>mean</th>
          <td>106015.860700</td>
          <td>185.425900</td>
          <td>25.967420</td>
          <td>5.591200</td>
          <td>3.120200</td>
          <td>79.364800</td>
          <td>47.773800</td>
          <td>33.860100</td>
          <td>15.775600</td>
          <td>4.147000</td>
          <td>6.944000</td>
          <td>0.504400</td>
        </tr>
        <tr>
          <th>std</th>
          <td>14172.725522</td>
          <td>6.581918</td>
          <td>3.818011</td>
          <td>5.026869</td>
          <td>2.534402</td>
          <td>29.402047</td>
          <td>19.260268</td>
          <td>14.237858</td>
```

```
          <td>7.057792</td>
          <td>3.248484</td>
          <td>4.443674</td>
          <td>0.500006</td>
        </tr>
        <tr>
          <th>min</th>
          <td>100284.000000</td>
          <td>168.000000</td>
          <td>16.200000</td>
          <td>0.000000</td>
          <td>0.000000</td>
          <td>0.000000</td>
          <td>0.000000</td>
          <td>0.000000</td>
          <td>0.000000</td>
          <td>0.000000</td>
          <td>0.000000</td>
          <td>0.000000</td>
        </tr>
        <tr>
          <th>25%</th>
          <td>102110.000000</td>
          <td>180.000000</td>
          <td>23.200000</td>
          <td>2.000000</td>
          <td>1.000000</td>
          <td>58.000000</td>
          <td>34.000000</td>
          <td>24.000000</td>
          <td>11.000000</td>
          <td>2.000000</td>
          <td>4.000000</td>
          <td>0.000000</td>
        </tr>
        <tr>
          <th>50%</th>
          <td>103507.000000</td>
          <td>185.000000</td>
          <td>25.700000</td>
          <td>4.000000</td>
          <td>3.000000</td>
          <td>74.000000</td>
          <td>44.000000</td>
          <td>31.000000</td>
          <td>15.000000</td>
          <td>4.000000</td>
          <td>6.000000</td>
          <td>1.000000</td>
```

```
        </tr>
        <tr>
          <th>75%</th>
          <td>104745.000000</td>
          <td>190.000000</td>
          <td>28.600000</td>
          <td>8.000000</td>
          <td>4.000000</td>
          <td>95.000000</td>
          <td>58.000000</td>
          <td>41.000000</td>
          <td>20.000000</td>
          <td>6.000000</td>
          <td>10.000000</td>
          <td>1.000000</td>
        </tr>
        <tr>
          <th>max</th>
          <td>210013.000000</td>
          <td>211.000000</td>
          <td>41.600000</td>
          <td>75.000000</td>
          <td>20.000000</td>
          <td>240.000000</td>
          <td>170.000000</td>
          <td>127.000000</td>
          <td>60.000000</td>
          <td>24.000000</td>
          <td>31.000000</td>
          <td>1.000000</td>
        </tr>
      </tbody>
    </table>
    </div>



.. code:: ipython3

    dataset.info()


.. parsed-literal::

    <class 'pandas.core.frame.DataFrame'>
    RangeIndex: 10000 entries, 0 to 9999
    Data columns (total 14 columns):
     #   Column   Non-Null Count  Dtype
    ---  ------   --------------  -----
```

```
     0   id        10000 non-null  int64
     1   name      10000 non-null  object
     2   surface   10000 non-null  object
     3   ht        10000 non-null  float64
     4   age       10000 non-null  float64
     5   ace       10000 non-null  float64
     6   df        10000 non-null  float64
     7   svpt      10000 non-null  float64
     8   1stIn     10000 non-null  float64
     9   1stWon    10000 non-null  float64
     10  2ndWon    10000 non-null  float64
     11  bpSaved   10000 non-null  float64
     12  bpFaced   10000 non-null  float64
     13  win       10000 non-null  int64
    dtypes: float64(10), int64(2), object(2)
    memory usage: 1.1+ MB


.. code:: ipython3


    numeric_cols = dataset.drop(columns=['id', 'name', 'win', 'surface']).columns

    # Create a 2x5 grid
    fig, axes = plt.subplots(nrows=5, ncols=2, figsize=(7, 10))
    axes = axes.flatten()  # Flatten the 2D array for easy iteration

    # Plot each histogram
    for i, col in enumerate(numeric_cols):
        sns.histplot(data=dataset, x=col, bins=50, hue="surface", hue_order = ['Hard','Clay','Grass','Carpet'], ax=axes[i],multiple="stack")
        axes[i].set_title(col)
        axes[i].grid(True)


    fig.suptitle("Distribution of Features by Surface Type", fontsize=16, fontweight='bold')
    plt.tight_layout()


    plt.savefig("Distribution of Features by Surface Type.png", format="png", dpi=500)
    plt.show()



.. image:: output_7_0.png
```

```


.. code:: ipython3

    le = LabelEncoder()
    dataset['surface_num'] = le.fit_transform(dataset['surface'])

.. code:: ipython3

    corr = dataset.drop(columns=['id', 'name', 'surface']).corr().round(2).reset_index().melt(id_vars="index", var_name="columns", value_name="value")

    corr = corr.rename(columns={"index": "rows"})

    plot = (ggplot(corr, aes(x='columns', y='rows', fill='value'))
            + geom_tile()
            + scale_fill_gradient2(low="blue", high="red", mid="white", midpoint=0)
            + geom_text(aes(label='value'), color='black', size=8)
            + theme_minimal()
            + labs(title="Correlation Matrix Heatmap")
           )

    #plot.tight_layout()
    plot.show()
    ggsave(plot, filename='correlation_matrix.png', width=7.5, height=5)



.. image:: output_9_0.png
   :width: 640px
   :height: 480px


.. parsed-literal::

    C:\Users\mmcdo\miniconda3\Lib\site-packages\plotnine\ggplot.py:615: PlotnineWarning: Saving 7.5 x 5 in image.
    C:\Users\mmcdo\miniconda3\Lib\site-packages\plotnine\ggplot.py:616: PlotnineWarning: Filename: correlation_matrix.png


.. code:: ipython3

    data = dataset.drop(columns=['id', 'name', 'surface'])
    response = data[['win']]
    features = data.drop(columns=['win'])
```

```
    train_size = round(len(data)*0.9)
    test_size = round(len(data)*0.1)

    train_features, test_features, train_response, test_response = train_test_split(
      features, response, train_size = train_size , test_size = test_size, random_state=123)


    train_data = pd.concat([train_response, train_features], axis=1)
    test_data = pd.concat([test_response, test_features], axis=1)

.. code:: ipython3

    def standardize_data(features,response):


      #scale dataset
      feautre_scaler = StandardScaler()
      response_scaler = StandardScaler()
      feautre_scaler.fit(features)
      response_scaler.fit(response)

      features_scaled = feautre_scaler.transform(features) #scaled Features
      response_scaled = response_scaler.transform(response) # scaled Response



      features_scaled = pd.DataFrame(features_scaled, columns=features.columns)
      response_scaled = pd.DataFrame(response_scaled, columns=response.columns)
      return features_scaled,response_scaled,feautre_scaler, response_scaler

.. code:: ipython3

    X_train = train_features
    Y_train = train_response
    X_test = test_features
    Y_test = test_response


    X_train_stand, Y_train_stand,X_train_stand_scaler, Y_train_stand_scaler = standardize_data(features=X_train, response=Y_train)
```

```
    X_test_stand, Y_test_stand, X_test_stand_scaler,
    Y_test_stand_scaler = standardize_data(features=X_test, response=
    Y_test)

    X_train_stand = sm.add_constant(X_train_stand)
    X_test_stand = sm.add_constant(X_test_stand)


    logit = sm.Logit(Y_train.reset_index(drop=True), X_train_stand)
    logit = logit.fit()

    # Print summary
    print(logit.summary())

    train_predictions = logit.predict(X_train_stand)
    test_predictions = logit.predict(X_test_stand)


    train_predictions = (train_predictions > 0.5).astype(int)
    test_predictions = (test_predictions > 0.5).astype(int)

    # lr = LogisticRegression()
    # lr.fit(X_train_scaled, y_train)
    # y_pred = lr.predict(X_test_scaled)

    # nb = GaussianNB(priors = [0.5,0.5])

    # nb.fit(X_train, Y_train.values.ravel())

    # train_predictions = nb.predict(X_train)
    # test_predictions = nb.predict(X_test)

    lr_train_err = 1 - accuracy_score(y_true=Y_train.values, y_pred=
    train_predictions)
    lr_test_err = 1 - accuracy_score(y_true=Y_test.values, y_pred=
    test_predictions)
    print(f"Train Error: {round(lr_train_err, 4) }")
    print(f"Test Error: {round(lr_test_err, 4) }")


.. parsed-literal::

    Optimization terminated successfully.
             Current function value: 0.453582
             Iterations 6
                               Logit Regression Results

    ==============================================================================
```

```
Dep. Variable:                      win   No. Observations:
       9000
Model:                            Logit   Df Residuals:
       8988
Method:                             MLE   Df Model:
         11
Date:                  Thu, 13 Mar 2025   Pseudo R-squ.:
     0.3456
Time:                          17:43:24   Log-Likelihood:
    -4082.2
converged:                         True   LL-Null:
    -6238.2
Covariance Type:              nonrobust   LLR p-value:
      0.000

==============================================================================
                  coef    std err          z      P>|z|
[0.025      0.975]

------------------------------------------------------------------------------
const          -0.0717      0.028     -2.580      0.010
-0.126      -0.017
ht             -0.2082      0.030     -6.854      0.000
-0.268      -0.149
age            -0.1853      0.028     -6.661      0.000
-0.240      -0.131
ace            -0.3076      0.039     -7.817      0.000
-0.385      -0.230
df             -0.0243      0.036     -0.676      0.499
-0.095       0.046
svpt           -3.2817      0.268    -12.260      0.000
-3.806      -2.757
1stIn          -0.1982      0.166     -1.193      0.233
-0.524       0.127
1stWon          2.9752      0.181     16.462      0.000
2.621       3.329
2ndWon          1.4224      0.096     14.856      0.000
1.235       1.610
bpSaved         1.7917      0.103     17.404      0.000
1.590       1.994
bpFaced        -2.2650      0.145    -15.580      0.000
-2.550      -1.980
surface_num    -0.1126      0.028     -4.049      0.000
-0.167      -0.058

==============================================================================
```

```
    Train Error: 0.2012
    Test Error: 0.206


.. code:: ipython3

    X_train = train_features
    Y_train = train_response
    X_test = test_features
    Y_test = test_response

    X_train_stand, Y_train_stand,X_train_stand_scaler, Y_train_stand_scaler = standardize_data(features=X_train, response=Y_train)
    X_test_stand, Y_test_stand, X_test_stand_scaler, Y_test_stand_scaler = standardize_data(features=X_test, response=Y_test)

    X_train_stand = sm.add_constant(X_train_stand)
    X_test_stand = sm.add_constant(X_test_stand)

    n1 = round(len(data)*0.9)
    n2 = round(len(data)*0.1)
    b = [10]
    # Initialize the result DataFrame
    result_df = pd.DataFrame({
        "B Cross Validations": [],
        "Average Error Rate": [],
        "Average Error Variance": []
    })

    for B in b:
        # Reinitialize the lists for storing metrics for each B

        test_error_rates = []
        test_accuracies = []

        for i in range(B):

            X_combined = pd.concat([X_train_stand, X_test_stand], axis=0)
            y_combined = pd.concat([Y_train, Y_test], axis=0).values.flatten()


            X_train_sample, X_test_sample, Y_train_sample, Y_test_sample = train_test_split(
                X_combined, y_combined, train_size=n1, test_size=n2, random_state=i
```

```
        )

        # Train the model on the training data
        logit = sm.Logit(Y_train_sample, X_train_sample)
        logit = logit.fit(disp=0)

        test_predictions = logit.predict(X_test_sample)



        test_predictions = (test_predictions > 0.5).astype(int)
        #lr.fit(X_train, Y_train)


        # Make predictions on the testing data




        # Calculate error rate (MSE)
        nb_test_error_rate = 1 - accuracy_score(y_true=Y_test_sample, y_pred=test_predictions)

        # Append to the lists

        test_error_rates.append(nb_test_error_rate)

    # Calculate the variance of the testing error rates after all iterations for this B
    test_error_variance = np.var(test_error_rates)

    # Calculate the averages for the current B value

    average_test_error_rate = np.mean(test_error_rates)


    metrics_df = pd.DataFrame({
        "B Cross Validations": [B],
        "Average Error Rate": [average_test_error_rate],
        "Average Error Variance": [test_error_variance]
    })


    result_df = pd.concat([result_df, metrics_df], axis=0)


result_df
```

```


.. raw:: html

    <div>
    <style scoped>
        .dataframe tbody tr th:only-of-type {
            vertical-align: middle;
        }

        .dataframe tbody tr th {
            vertical-align: top;
        }

        .dataframe thead th {
            text-align: right;
        }
    </style>
    <table border="1" class="dataframe">
      <thead>
        <tr style="text-align: right;">
          <th></th>
          <th>B Cross Validations</th>
          <th>Average Error Rate</th>
          <th>Average Error Variance</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <th>0</th>
          <td>10.0</td>
          <td>0.2083</td>
          <td>0.000179</td>
        </tr>
      </tbody>
    </table>
    </div>



.. code:: ipython3

    final_cv_result = pd.concat([pd.DataFrame({
        'Model': ['Logistic Regression'],
        'Mean Test Error': [result_df["Average Error Rate"].iloc[0]],
        'Test Error Var': [result_df["Average Error Variance"].iloc
    [0]]
    })])
    #model_errors['Logistic Regression'] = test_error_rates
```

```
.. code:: ipython3

    X_train = train_features
    Y_train = train_response
    X_test = test_features
    Y_test = test_response


    X_train_stand, Y_train_stand,X_train_stand_scaler, Y_train_stand_scaler = standardize_data(features=X_train, response=Y_train)
    X_test_stand, Y_test_stand, X_test_stand_scaler, Y_test_stand_scaler = standardize_data(features=X_test, response=Y_test)


    K = 101
    train_error = []
    test_error = []
    for k in range(1,K+1,5):
      knn = KNeighborsClassifier(n_neighbors=k)
      knn.fit(X_train_stand, Y_train.values.ravel())
      train_predictions = knn.predict(X_train_stand)
      test_predictions = knn.predict(X_test_stand)

      cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)
      train_score = 1 - model_selection.cross_val_score(knn, X_train_stand, Y_train.astype(int).values.ravel(), cv=cv, scoring='accuracy').mean()
      test_score = 1 - model_selection.cross_val_score(knn, X_test_stand, Y_test.astype(int).values.ravel(), cv=cv, scoring='accuracy').mean()

      # train_error.append(1 - accuracy_score(y_true=Y_train.values.ravel(), y_pred=train_predictions))
      train_error.append(train_score)
      # test_error.append(1 - accuracy_score(y_true=Y_test.values.ravel(), y_pred=test_predictions))
      test_error.append(test_score)


      # # Create a plot with dual y-axes
    plt.close('all')
    fig, ax1 = plt.subplots(figsize=(8, 6))

    # Plot MSE on the first y-axis
    ax1.plot(range(1,K+1,5), train_error, 'b-', marker='o', label='CV (K-fold) Training Error')
```

```
    ax1.set_xlabel('Number of Neighbors (k)')
    ax1.set_ylabel('Error', color='blue')
    ax1.tick_params(axis='y', labelcolor='blue')


    ax1.plot(range(1,K+1,5), test_error, 'g-', marker='o', label='CV (
   K-fold) Test Error')
    ax1.set_xlabel('Number of Neighbors (k)')
    #ax1.set_ylabel('Error', color='blue')
    ax1.tick_params(axis='y', labelcolor='blue')
    plt.legend()
    plt.grid()
    plt.title("K-Neighbors vs. Error")
    plt.savefig("K-Neighbors vs. Error.png", format="png", dpi=300)

    # knn_train_err = (train_error[12])
    # knn_test_err = (test_error[12])
    # print(f"Train Error: {round(knn_train_err, 4) }")
    # print(f"Test Error: {round(knn_test_err, 4) }")



.. image:: output_15_0.png


.. code:: ipython3

    knn_train_err = (train_error[12])
    knn_test_err = (test_error[12])
    print(f"Train Error: {round(knn_train_err, 4) }")
    print(f"Test Error: {round(knn_test_err, 4) }")


.. parsed-literal::

    Train Error: 0.2593
    Test Error: 0.277


.. code:: ipython3

    X_train = train_features
    Y_train = train_response
    X_test = test_features
    Y_test = test_response

    X_train_stand, Y_train_stand,X_train_stand_scaler,
   Y_train_stand_scaler = standardize_data(features=X_train, response
   =Y_train)
```

```
    X_test_stand, Y_test_stand, X_test_stand_scaler,
Y_test_stand_scaler = standardize_data(features=X_test, response=
Y_test)


    n1 = round(len(data)*0.9)
    n2 = round(len(data)*0.1)
    b = [10]
    # Initialize the result DataFrame
    result_df = pd.DataFrame({
        "B Cross Validations": [],
        "Average Error Rate": [],
        "Average Error Variance": []
    })

    for B in b:
        # Reinitialize the lists for storing metrics for each B

        test_error_rates = []
        test_accuracies = []

        for i in range(B):

            X_combined = pd.concat([X_train_stand, X_test_stand], axis
=0)
            y_combined = pd.concat([Y_train, Y_test], axis=0).values.
flatten()


            X_train_sample, X_test_sample, Y_train_sample,
Y_test_sample = train_test_split(
                X_combined, y_combined, train_size=n1, test_size=n2,
random_state=i
            )

            # Train the model on the training data
            knn = KNeighborsClassifier(n_neighbors=60)
            knn.fit(X_train_stand, Y_train.values.ravel())

            test_predictions = knn.predict(X_test_sample)



            test_predictions = (test_predictions > 0.5).astype(int)
            #lr.fit(X_train, Y_train)


            # Make predictions on the testing data

```

```



            # Calculate error rate (MSE)
            nb_test_error_rate = 1 - accuracy_score(y_true=
    Y_test_sample, y_pred=test_predictions)

            # Append to the lists

            test_error_rates.append(nb_test_error_rate)

        # Calculate the variance of the testing error rates after all
    iterations for this B
        test_error_variance = np.var(test_error_rates)

        # Calculate the averages for the current B value

        average_test_error_rate = np.mean(test_error_rates)


        metrics_df = pd.DataFrame({
            "B Cross Validations": [B],
            "Average Error Rate": [average_test_error_rate],
            "Average Error Variance": [test_error_variance]
        })


        result_df = pd.concat([result_df, metrics_df], axis=0)


    result_df




.. raw:: html

    <div>
    <style scoped>
        .dataframe tbody tr th:only-of-type {
            vertical-align: middle;
        }

        .dataframe tbody tr th {
            vertical-align: top;
        }

        .dataframe thead th {
            text-align: right;
```

```
        }
    </style>
    <table border="1" class="dataframe">
      <thead>
        <tr style="text-align: right;">
          <th></th>
          <th>B Cross Validations</th>
          <th>Average Error Rate</th>
          <th>Average Error Variance</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <th>0</th>
          <td>10.0</td>
          <td>0.2497</td>
          <td>0.000112</td>
        </tr>
      </tbody>
    </table>
    </div>



.. code:: ipython3

    final_cv_result = pd.concat([final_cv_result,pd.DataFrame({
        'Model': ['KNN'],
        'Mean Test Error': [result_df["Average Error Rate"].iloc[0]],
        'Test Error Var': [result_df["Average Error Variance"].iloc
    [0]]
    })])

.. code:: ipython3

    X_train = train_features
    Y_train = train_response
    X_test = test_features
    Y_test = test_response

    # Standardize Data (Assuming standardize_data() function works as
    expected)
    X_train_stand, Y_train_stand, X_train_stand_scaler,
    Y_train_stand_scaler = standardize_data(features=X_train, response
    =Y_train)
    X_test_stand, Y_test_stand, X_test_stand_scaler,
    Y_test_stand_scaler = standardize_data(features=X_test, response=
    Y_test)

```

```
    Y_train_stand = Y_train.values.ravel() # Ensure Y values remain binary (0/1)
    Y_test_stand = Y_test.values.ravel()

    # Define range of estimators (1 to 101 in steps of 5)
    n_estimators_values = list(range(1, 101, 5))

    # Lists to store errors
    train_error = []
    test_error = []

    # Cross-validation setup
    cv = RepeatedKFold(n_splits=10, n_repeats=3, random_state=1)

    # Loop through different n_estimators values
    for n_estimators in tqdm(n_estimators_values):
        rf_model = RandomForestClassifier(n_estimators=n_estimators, max_depth=3, random_state=42)

        # Fit the model
        rf_model.fit(X_train_stand, Y_train_stand)

        # Cross-validation scores
        train_score = 1 - cross_val_score(rf_model, X_train_stand, Y_train_stand, cv=cv, scoring='accuracy').mean()
        test_score = 1 - cross_val_score(rf_model, X_test_stand, Y_test_stand, cv=cv, scoring='accuracy').mean()

        # Append errors
        train_error.append(train_score)
        test_error.append(test_score)

    # Plot results
    plt.figure(figsize=(8, 6))
    plt.plot(n_estimators_values, train_error, 'b-', marker='o', label='CV (K-fold) Training Error')
    plt.plot(n_estimators_values, test_error, 'g-', marker='o', label='CV (K-fold) Test Error')
    plt.xlabel('Number of Estimators')
    plt.ylabel('Error')
    plt.title("Random Forest: Number of Estimators vs. Error")
    plt.legend()
    plt.grid()
    plt.savefig("RandomForest_NumEstimators_vs_Error.png", format="png", dpi=300)
    plt.show()


.. parsed-literal::
```

```
.. image:: output_19_1.png


.. code:: ipython3

    X_train = train_features
    Y_train = train_response
    X_test = test_features
    Y_test = test_response

    X_train_stand, Y_train_stand,X_train_stand_scaler, Y_train_stand_scaler = standardize_data(features=X_train, response=Y_train)
    X_test_stand, Y_test_stand, X_test_stand_scaler, Y_test_stand_scaler = standardize_data(features=X_test, response=Y_test)


    Y_train_stand = Y_train.values.ravel() # Ensure Y values remain binary (0/1)
    Y_test_stand = Y_test.values.ravel()

    n1 = round(len(data)*0.9)
    n2 = round(len(data)*0.1)
    b = [10]
    # Initialize the result DataFrame
    result_df = pd.DataFrame({
        "B Cross Validations": [],
        "Average Error Rate": [],
        "Average Error Variance": []
    })

    for B in b:
        # Reinitialize the lists for storing metrics for each B

        test_error_rates = []
        test_accuracies = []

        for i in range(B):

            X_combined = pd.concat([X_train_stand, X_test_stand], axis=0)
            y_combined = pd.concat([Y_train, Y_test], axis=0).values.flatten()
```

```
        X_train_sample, X_test_sample, Y_train_sample, Y_test_sample = train_test_split(
            X_combined, y_combined, train_size=n1, test_size=n2, random_state=i
        )

        # Train the model on the training data
        rf_model = RandomForestClassifier(n_estimators=80, max_depth=3, random_state=42)

        # Fit the model
        rf_model.fit(X_train_stand, Y_train_stand)

        test_predictions = rf_model.predict(X_test_sample)



        test_predictions = (test_predictions > 0.5).astype(int)
        #lr.fit(X_train, Y_train)


        # Make predictions on the testing data




        # Calculate error rate (MSE)
        nb_test_error_rate = 1 - accuracy_score(y_true=Y_test_sample, y_pred=test_predictions)

        # Append to the lists

        test_error_rates.append(nb_test_error_rate)

    # Calculate the variance of the testing error rates after all iterations for this B
    test_error_variance = np.var(test_error_rates)

    # Calculate the averages for the current B value

    average_test_error_rate = np.mean(test_error_rates)


    metrics_df = pd.DataFrame({
        "B Cross Validations": [B],
        "Average Error Rate": [average_test_error_rate],
        "Average Error Variance": [test_error_variance]
    })

```

```

        result_df = pd.concat([result_df, metrics_df], axis=0)


    result_df




.. raw:: html

    <div>
    <style scoped>
        .dataframe tbody tr th:only-of-type {
            vertical-align: middle;
        }

        .dataframe tbody tr th {
            vertical-align: top;
        }

        .dataframe thead th {
            text-align: right;
        }
    </style>
    <table border="1" class="dataframe">
      <thead>
        <tr style="text-align: right;">
          <th></th>
          <th>B Cross Validations</th>
          <th>Average Error Rate</th>
          <th>Average Error Variance</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <th>0</th>
          <td>10.0</td>
          <td>0.2869</td>
          <td>0.000169</td>
        </tr>
      </tbody>
    </table>
    </div>



.. code:: ipython3

```

```
    final_cv_result = pd.concat([final_cv_result,pd.DataFrame({
        'Model': ['Random Forest'],
        'Mean Test Error': [result_df["Average Error Rate"].iloc[0]],
        'Test Error Var': [result_df["Average Error Variance"].iloc
[0]]
    })])

.. code:: ipython3

    X_train = train_features
    Y_train = train_response
    X_test = test_features
    Y_test = test_response

    # Standardize Data (Assuming standardize_data() function works as
expected)
    X_train_stand, Y_train_stand, X_train_stand_scaler,
Y_train_stand_scaler = standardize_data(features=X_train, response
=Y_train)
    X_test_stand, Y_test_stand, X_test_stand_scaler,
Y_test_stand_scaler = standardize_data(features=X_test, response=
Y_test)

    Y_train_stand = Y_train
    Y_test_stand = Y_test

    # Define range of learning rates (eta)
    eta_values = np.linspace(0.01, 0.36, 25)  # From 0.01 to 0.3 in 10
 steps

    # Lists to store errors
    train_error = []
    test_error = []

    # Cross-validation setup
    cv = RepeatedKFold(n_splits=10, n_repeats=5, random_state=42)

    # Loop through different eta values
    for eta in tqdm(eta_values):
        xgb_model = xgb.XGBClassifier(learning_rate=eta, n_estimators
=80, max_depth=3, random_state=42, eval_metric="error")

        # Fit the model
        xgb_model.fit(X_train_stand, Y_train_stand)

        # Cross-validation scores
        train_score = 1 - cross_val_score(xgb_model, X_train_stand,
Y_train_stand, cv=cv, scoring='accuracy').mean()
```

```
        test_score = 1 - cross_val_score(xgb_model, X_test_stand, Y_test_stand, cv=cv, scoring='accuracy').mean()

        # Append errors
        train_error.append(train_score)
        test_error.append(test_score)

    # Plot results
    plt.figure(figsize=(8, 6))
    plt.plot(eta_values, train_error, 'b-', marker='o', label='CV (K-fold) Training Error')
    plt.plot(eta_values, test_error, 'g-', marker='o', label='CV (K-fold) Test Error')
    plt.xlabel('Learning Rate (eta)')
    plt.ylabel('Error')
    plt.title("XGBoost Learning Rate vs. Error")
    plt.legend()
    plt.grid()
    plt.savefig("XGBoost_LearningRate_vs_Error.png", format="png", dpi=300)
    plt.show()


.. parsed-literal::





.. image:: output_22_1.png


.. code:: ipython3

    xgb_model = xgb.XGBClassifier(learning_rate=.21, n_estimators=80, max_depth=3, random_state=42, eval_metric="error")

    # Fit the model
    xgb_model.fit(X_train_stand, Y_train_stand)

    # Cross-validation scores
    train_score = 1 - cross_val_score(xgb_model, X_train_stand, Y_train_stand, cv=cv, scoring='accuracy').mean()
    test_score = 1 - cross_val_score(xgb_model, X_test_stand, Y_test_stand, cv=cv, scoring='accuracy').mean()

    print("train: : " ,train_score)
    print("test: : " ,test_score)
```

```

.. parsed-literal::

    train: :  0.21211111111111114
    test: :  0.24760000000000015


.. code:: ipython3

    X_train = train_features
    Y_train = train_response
    X_test = test_features
    Y_test = test_response

    X_train_stand, Y_train_stand,X_train_stand_scaler, Y_train_stand_scaler = standardize_data(features=X_train, response=Y_train)
    X_test_stand, Y_test_stand, X_test_stand_scaler, Y_test_stand_scaler = standardize_data(features=X_test, response=Y_test)


    Y_train_stand = Y_train# Ensure Y values remain binary (0/1)
    Y_test_stand = Y_test

    n1 = round(len(data)*0.9)
    n2 = round(len(data)*0.1)
    b = [100]
    # Initialize the result DataFrame
    result_df = pd.DataFrame({
        "B Cross Validations": [],
        "Average Error Rate": [],
        "Average Error Variance": []
    })

    for B in tqdm(b):
        # Reinitialize the lists for storing metrics for each B

        test_error_rates = []
        test_accuracies = []

        for i in range(B):

            X_combined = pd.concat([X_train_stand, X_test_stand], axis=0)
            y_combined = pd.concat([Y_train, Y_test], axis=0).values.flatten()


```

```python
        X_train_sample, X_test_sample, Y_train_sample, Y_test_sample = train_test_split(
            X_combined, y_combined, train_size=n1, test_size=n2, random_state=i
        )

        # Train the model on the training data
        xgb_model = xgb.XGBClassifier(learning_rate=.21, n_estimators=80, max_depth=3, random_state=42, eval_metric="error")

        # Fit the model
        xgb_model.fit(X_train_stand, Y_train_stand)

        test_predictions = xgb_model.predict(X_test_sample)



        test_predictions = (test_predictions > 0.5).astype(int)
        #lr.fit(X_train, Y_train)


        # Make predictions on the testing data




        # Calculate error rate (MSE)
        nb_test_error_rate = 1 - accuracy_score(y_true=Y_test_sample, y_pred=test_predictions)

        # Append to the lists

        test_error_rates.append(nb_test_error_rate)

    # Calculate the variance of the testing error rates after all iterations for this B
    test_error_variance = np.var(test_error_rates)

    # Calculate the averages for the current B value

    average_test_error_rate = np.mean(test_error_rates)


    metrics_df = pd.DataFrame({
        "B Cross Validations": [B],
        "Average Error Rate": [average_test_error_rate],
        "Average Error Variance": [test_error_variance]
    })
```

```


        result_df = pd.concat([result_df, metrics_df], axis=0)


    result_df


.. parsed-literal::






.. raw:: html

    <div>
    <style scoped>
        .dataframe tbody tr th:only-of-type {
            vertical-align: middle;
        }

        .dataframe tbody tr th {
            vertical-align: top;
        }

        .dataframe thead th {
            text-align: right;
        }
    </style>
    <table border="1" class="dataframe">
      <thead>
        <tr style="text-align: right;">
          <th></th>
          <th>B Cross Validations</th>
          <th>Average Error Rate</th>
          <th>Average Error Variance</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <th>0</th>
          <td>100.0</td>
          <td>0.18749</td>
          <td>0.00016</td>
        </tr>
      </tbody>
    </table>
```

```
    </div>



.. code:: ipython3

    final_cv_result = pd.concat([final_cv_result,pd.DataFrame({
        'Model': ['XGBoost'],
        'Mean Test Error': [result_df["Average Error Rate"].iloc[0]],
        'Test Error Var': [result_df["Average Error Variance"].iloc[0]]
    })])

.. code:: ipython3

    final_cv_result




.. raw:: html

    <div>
    <style scoped>
        .dataframe tbody tr th:only-of-type {
            vertical-align: middle;
        }

        .dataframe tbody tr th {
            vertical-align: top;
        }

        .dataframe thead th {
            text-align: right;
        }
    </style>
    <table border="1" class="dataframe">
      <thead>
        <tr style="text-align: right;">
          <th></th>
          <th>Model</th>
          <th>Mean Test Error</th>
          <th>Test Error Var</th>
        </tr>
      </thead>
      <tbody>
        <tr>
          <th>0</th>
          <td>Logistic Regression</td>
```

```
          <td>0.20830</td>
          <td>0.000179</td>
        </tr>
        <tr>
          <th>0</th>
          <td>KNN</td>
          <td>0.24970</td>
          <td>0.000112</td>
        </tr>
        <tr>
          <th>0</th>
          <td>Random Forest</td>
          <td>0.28690</td>
          <td>0.000169</td>
        </tr>
        <tr>
          <th>0</th>
          <td>XGBoost</td>
          <td>0.18749</td>
          <td>0.000160</td>
        </tr>
      </tbody>
    </table>
    </div>



.. code:: ipython3

    # X_train = train_features
    # Y_train = train_response
    # X_test = test_features
    # Y_test = test_response

    # #model = LinearRegression()

    # sfs = SequentialFeatureSelector(estimator = linear_model.LinearRegression(),
    #                                  k_features='best',
    #                                  forward=True,
    #                                  floating = True,
    #                                  scoring=aic_score,
    #                                  cv=None)
    # selected_features = sfs.fit(X_train, Y_train)
    # selected_features = list(sfs.k_feature_names_)

    # X_train_stepwise = X_train[selected_features]
    # X_test_stepwise = X_test[selected_features]

```

```
    # X_train_stepwise = sm.add_constant(X_train_stepwise)
    # X_test_stepwise = sm.add_constant(X_test_stepwise)

    # lr_stepwise = sm.OLS(Y_train, X_train_stepwise).fit()

    # train_predictions = lr_stepwise.predict(X_train_stepwise)
    # test_predictions = lr_stepwise.predict(X_test_stepwise)


    # lr_stepwise_train_mse = mean_squared_error(y_true=Y_train.values
    , y_pred=train_predictions)
    # lr_stepwise_test_mse = mean_squared_error(y_true=Y_test.values,
    y_pred=test_predictions)

    # print(lr_stepwise.summary())

    # print(f"Train MSE: {round(lr_stepwise_train_mse, 4) }")
    # print(f"Test MSE: {round(lr_stepwise_test_mse, 4) }")

.. code:: ipython3

    Y_test_probs = logit.predict(X_test_stand)  # Use probability
    scores

    # Compute ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(Y_test, Y_test_probs)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (
    AUC = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label
    ='Chance (AUC = 0.50)')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Logistic Regression ROC Curve')
    plt.legend(loc="lower right")

    # Save and show plot
    plt.grid(True)
    plt.savefig("roc_curve.png", format="png", dpi=300)
    plt.show()

    knn = KNeighborsClassifier(n_neighbors=60)
    knn.fit(X_train_stand, Y_train.values.ravel())
    Y_test_probs = knn.predict(X_test_stand)
```

```
    # Compute ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(Y_test, Y_test_probs)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Chance (AUC = 0.50)')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('KNN ROC Curve')
    plt.legend(loc="lower right")

    # Save and show plot
    plt.grid(True)
    plt.savefig("roc_curve.png", format="png", dpi=300)
    plt.show()
    rf_model = RandomForestClassifier(n_estimators=80, max_depth=3, random_state=42)

    # Fit the model
    rf_model.fit(X_train_stand, Y_train_stand)
    Y_test_probs = rf_model.predict(X_test_stand)


    # Compute ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(Y_test, Y_test_probs)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label='Chance (AUC = 0.50)')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('Random Forrest ROC Curve')
    plt.legend(loc="lower right")

    # Save and show plot
    plt.grid(True)
    plt.savefig("roc_curve.png", format="png", dpi=300)
    plt.show()
    xgb_model = xgb.XGBClassifier(learning_rate=.21, n_estimators=80, max_depth=3, random_state=42, eval_metric="error")

```

```
    # Fit the model
    xgb_model.fit(X_train_stand, Y_train_stand)
    Y_test_probs = xgb_model.predict(X_test_stand)


    # Compute ROC curve and AUC
    fpr, tpr, thresholds = roc_curve(Y_test, Y_test_probs)
    roc_auc = auc(fpr, tpr)

    # Plot ROC curve
    plt.figure(figsize=(8, 6))
    plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (
   AUC = {roc_auc:.2f})')
    plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--', label
   ='Chance (AUC = 0.50)')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.title('XGBoost ROC Curve')
    plt.legend(loc="lower right")

    # Save and show plot
    plt.grid(True)
    plt.savefig("roc_curve.png", format="png", dpi=300)
    plt.show()


::



   ---------------------------------------------------------------------------


    ValueError                                Traceback (most recent
   call last)

    Cell In[28], line 1
    ----> 1 Y_test_probs = logit.predict(X_test_stand)  # Use
   probability scores
          3 # Compute ROC curve and AUC
          4 fpr, tpr, thresholds = roc_curve(Y_test, Y_test_probs)


    File ~\miniconda3\Lib\site-packages\statsmodels\base\model.py
   :1174, in Results.predict(self, exog, transform, *args, **kwargs)
       1127 """
       1128 Call self.model.predict with self.params as the first
   argument.
       1129
```

```
    (...)
    1169 returned prediction.
    1170 """
    1171 exog, exog_index = self._transform_predict_exog(exog,
    1172                                                         transform=
transform)
 -> 1174 predict_results = self.model.predict(self.params, exog, *
args,
    1175                                         **kwargs)
    1177 if exog_index is not None and not hasattr(predict_results,
    1178                                         '
predicted_values'):
    1179     if predict_results.ndim == 1:


 File ~\miniconda3\Lib\site-packages\statsmodels\discrete\
discrete_model.py:543, in BinaryModel.predict(self, params, exog,
which, linear, offset)
     540 if exog is None:
     541     exog = self.exog
 --> 543 linpred = np.dot(exog, params) + offset
     545 if which == "mean":
     546     return self.cdf(linpred)


 ValueError: shapes (1000,11) and (12,) not aligned: 11 (dim 1) !=
12 (dim 0)
```
Listing 1: Code