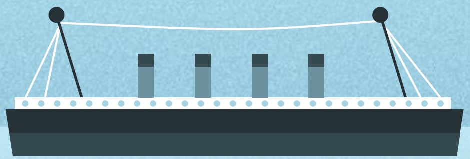
— Titanic data set analysis—

Iart proj2

Up202108689 – António Azevedo
Up202108794 – José Martins
Up202108776 – Tomás Martins



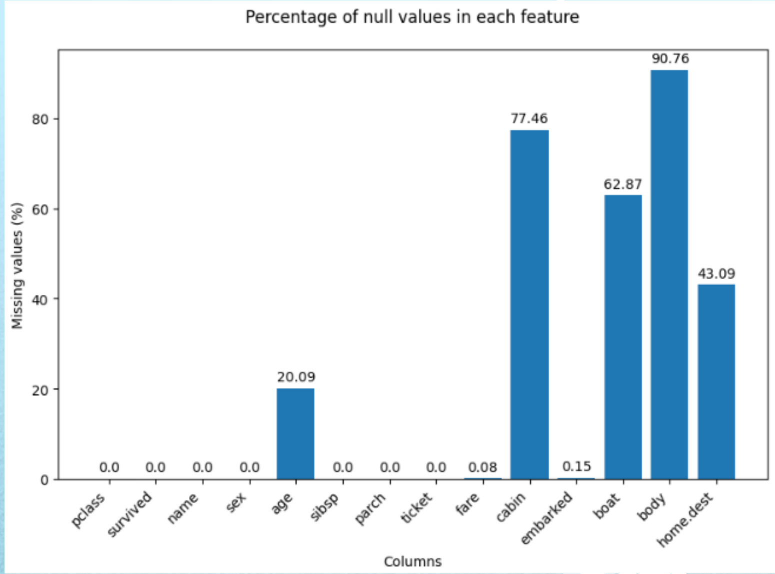
— The data set features —

1. Passenger class;
2. Survived;
3. Home destination;
4. Name;
5. Sex;
6. Age;
7. Sibling/Spouse;
8. Parent/Children;
9. Ticket;
10. Fare;
11. Cabin;
12. Embarked;
13. Boat;
14. Body;

— The analysis—

Data cleaning and goal

To predict whether an individual survived the Titanic disaster using machine learning, we needed to clean our dataset. We removed features with a high percentage of missing values and handled other missing values by various methods, such as replacing them with the column's mean.



— Algorithms—

Logistic Regression	Extra Trees	Multinomial Naive Bayes
Random Forest	SVM	KNN
Decision Tress	Bernoulli Naive Bayes	Gradient Boosting
MLP	Gaussian Naive Bayes	

— Tools—

Pandas

“Pandas is a Python library used for (…) analyzing, cleaning, exploring, and manipulating data.”

Scikit-Learn

“Scikit-learn is a library in Python that provides many unsupervised and supervised learning algorithms.”

Seaborn

“Seaborn is a Python data visualization library based on matplotlib. It provides a high-level interface for drawing attractive and informative statistical graphics.”

Matplot

“Matplotlib is a comprehensive library for creating static, animated, and interactive visualizations in Python.”

— Train-test split—

Upon concluding that our dataset was significantly unbalanced, we evaluated various balancing techniques: stratification, oversampling, and undersampling. We used accuracy as our metric due to being sufficiently descriptive and easy to use. Although undersampling gave the highest accuracy, it risked substantial information loss by reducing the dataset size. Consequently, we opted for stratification, which preserves the survival ratio between the training and test datasets, ensuring a more robust model without sacrificing data integrity.

Then we split the data into 80% testing and 20% training.

Sampling Technique	AllKNN	SMOTE	Stratified
Model			
Bernoulli Naive Bayes	0.790850	0.759398	0.801527
Decision Tree	0.875817	0.774436	0.774809
Extra Trees	0.895425	0.800752	0.778626
Gaussian Naive Bayes	0.836601	0.759398	0.786260
Gradient Boosting	0.915033	0.838346	0.839695
KNN	0.790850	0.751880	0.702290
Logistic Regression	0.856209	0.751880	0.816794
MLP	0.869281	0.733083	0.816794
Multinomial Naive Bayes	0.718954	0.639098	0.675573
Random Forest	0.908497	0.796992	0.797710
SVM	0.686275	0.646617	0.667939

	Accuracy
Sampling Technique	
AllKNN	0.831254
Stratified	0.768910
SMOTE	0.750171

```
Number of samples in the training set after undersampling:
{0: 306, 1: 303}

Number of samples in the testing set after undersampling:
{0: 74, 1: 79}

Total number of samples post undersampling:  762

Total number of samples in the original data:  1309
```

— Model testing and parameters tuning—

For each algorithm we tested, we generated a classification report that included accuracy, precision, recall, F1-score, and support metrics. To maximize each algorithm's performance, we used GridSearchCV to find the optimal hyperparameters, aiming for the best overall results.

Finally, we compared the performance of the tuned models to those with default parameters, highlighting the improvements achieved through hyperparameter optimization.

```python
param_grid = [
    {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000], 'penalty': ['l1'], 'solver': ['liblinear']},
    {'C': [0.001, 0.01, 0.1, 1, 10, 100, 1000], 'penalty': ['l2'], 'solver': ['liblinear', 'lbfgs']}
]

# Create a Logistic Regression model
log_reg = LogisticRegression(max_iter=1000, random_state=42)

# Create the grid search object
grid_search = GridSearchCV(log_reg, param_grid, cv=3, verbose=0, error_score=np.nan)

# Fit the grid search
grid_search.fit(X_trainStrt, y_trainStrt)

# Get the best parameters and best model
best_params = grid_search.best_params_

print(f"The best parameters for Logistic Regression are: {best_params}")
```

```
The best parameters for Logistic Regression are: {'C': 0.1, 'penalty': 'l2', 'solver': 'lbfgs'}
```

```python
modelTune = LogisticRegression(max_iter=1000, C=0.1, penalty='l2', solver='lbfgs', random_state=42)
modelTune.fit(X_trainStrt, y_trainStrt)

# Make predictions
y_pred = modelTune.predict(X_testStrt)

# Print classification report
print(classification_report(y_testStrt, y_pred))
print("Accuracy: ", accuracy_score(y_testStrt, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.84      0.89      0.86       162
           1       0.80      0.73      0.76       100

    accuracy                           0.83       262
   macro avg       0.82      0.81      0.81       262
weighted avg       0.83      0.83      0.83       262

Accuracy:  0.8282442748091603
```

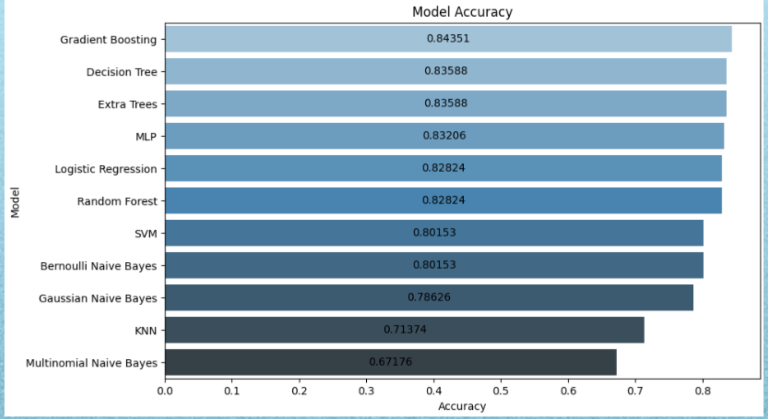
```python
modelDefault = LogisticRegression(max_iter=1000, random_state=42)
modelDefault.fit(X_trainStrt, y_trainStrt)

# Make predictions
y_pred = modelDefault.predict(X_testStrt)

# Print classification report
print(classification_report(y_testStrt, y_pred))
print("Accuracy: ", accuracy_score(y_testStrt, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.84      0.87      0.85       162
           1       0.78      0.73      0.75       100

    accuracy                           0.82       262
   macro avg       0.81      0.80      0.80       262
weighted avg       0.82      0.82      0.82       262

Accuracy:  0.816793893129771
```

— Model's results comparison—

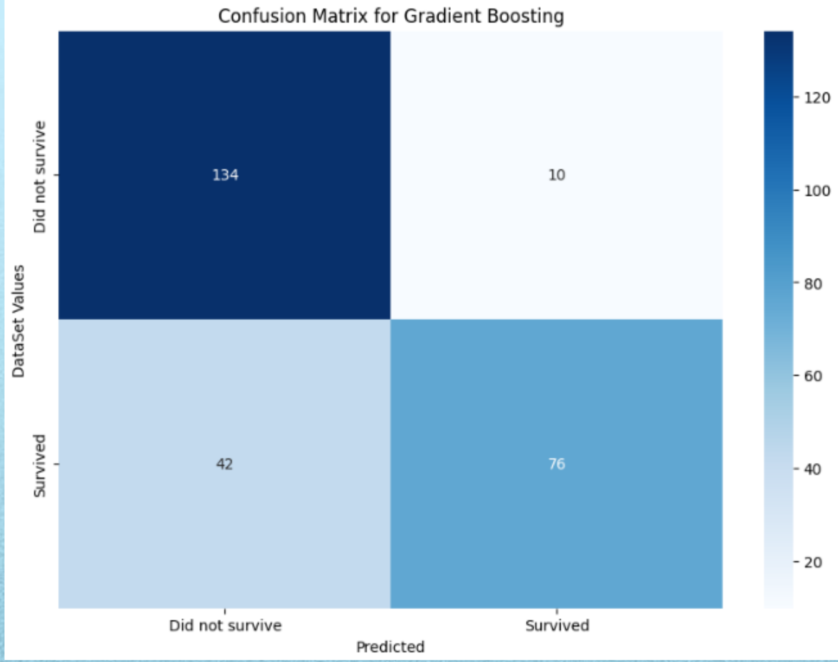
After identifying the optimal parameters for each model, we compared their performance to determine the best model for our dataset. We ran all the algorithms and compiled their results into a table, sorted by accuracy. To enhance visual clarity, we created a bar plot displaying the accuracy of each algorithm. This visual representation facilitated the comparison between models, even though accuracy alone may not fully capture all aspects of model effectiveness. Despite its limitations, accuracy provided a sufficiently descriptive overview to highlight the relative performance of each algorithm.

	Model	Accuracy	Precision	Recall	F1 Score
1	Gradient Boosting	0.843511	0.844643	0.817963	0.840144
2	Decision Tree	0.835878	0.834684	0.817531	0.834229
3	Extra Trees	0.835878	0.836590	0.809877	0.832346
4	MLP	0.832061	0.831104	0.810617	0.829700
5	Logistic Regression	0.828244	0.826873	0.809444	0.826519
6	Random Forest	0.828244	0.830523	0.797963	0.823397
7	SVM	0.801527	0.799876	0.784012	0.800262
8	Bernoulli Naive Bayes	0.801527	0.799876	0.784012	0.800262
9	Gaussian Naive Bayes	0.786260	0.785506	0.771667	0.785836
10	KNN	0.713740	0.708019	0.669012	0.701226
11	Multinomial Naive Bayes	0.671756	0.661950	0.631235	0.661906



— Model's results comparison—

Following this initial analysis, we conducted a deeper investigation into the results of the top-performing model, which was Gradient Boosting. To gain further insights, we generated a confusion matrix for this model. This allowed us to better understand its performance by examining the distribution of true positives, false positives, true negatives, and false negatives, providing a more detailed evaluation of its predictive capabilities.



—Sources—

LINKS

- Titanic Dataset – Kaggle;
- What is a Linear Regression – IBM;
- What is a Decision Tree – IBM;
- Random Forest – GeeksForGeeks;
- What is a Neural Network – IBM;
- Pandas – W3School;
- Seaborn – Seaborn;
- Scikit-learn – Codecademy;
- Matplot – Matplotlib;