Machine Learning Review

Jacques Nahri

June 26, 2025

Abstract

This report explores the application of machine learning algorithms on two well-known datasets: the Titanic and the Iris datasets. The objective is to demonstrate how data can be cleaned, processed, and modeled to make meaningful predictions. Logistic Regression is used for binary classification in the Titanic dataset, while K-Nearest Neighbors (KNN) is applied for multiclass classification in the Iris dataset. Performance metrics such as accuracy, confusion matrices, and classification reports are used to evaluate the models. The insights gained provide a foundation for understanding how similar data-driven techniques can be applied in real-world scenarios such as user behavior prediction and recommendation systems—especially in the context of platforms like 3VO.

Contents

1 Introduction

Machine learning has become an essential tool for extracting insights and making predictions based on data. To build and evaluate effective models, well-known datasets are often used for experimentation and learning. In this report, we explore two classic datasets: the Titanic dataset and the Iris dataset.

The Titanic dataset is commonly used for binary classification problems, where the goal is to predict whether a passenger survived the infamous Titanic disaster based on features such as age, sex, class, and fare. This dataset provides an excellent opportunity to practice data preprocessing, feature engineering, and evaluating model performance using classification metrics.

The Iris dataset, on the other hand, is one of the earliest and most widely used datasets for multiclass classification. It includes measurements of three types of Iris flowers—Setosa, Versicolor, and Virginica—based on petal and sepal dimensions. This dataset is ideal for testing classification algorithms like K-Nearest Neighbors (KNN) and understanding how models can distinguish between multiple classes.

By working with both datasets, we demonstrate how machine learning models can be applied to diverse problems, from binary survival prediction to multiclass species classification.

Dataset Description

The Titanic dataset includes the following columns:

- **PassengerId**: Unique identifier for each passenger.
- **Survived**: Survival (0 = No, 1 = Yes).
- **Pclass**: Passenger class (1 = 1st, 2 = 2nd, 3 = 3rd).
- **Name**: Full name of the passenger.
- **Sex**: Gender of the passenger.
- **Age**: Age of the passenger in years.
- **SibSp**: Number of siblings/spouses aboard.
- **Parch**: Number of parents/children aboard.
- **Ticket**: Ticket number.
- **Fare**: Fare paid.
- **Cabin**: Cabin number.
- **Embarked**: Port of Embarkation (C = Cherbourg, Q = Queenstown, S = Southampton).

2 Implementation

2.1 Importing Required Libraries

```python
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
import sklearn.metrics
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

Listing 1: Import Libraries

2.2 Reading the Titanic Dataset

```python
df = pd.read_csv('Titanic-Dataset.csv')
```

Listing 2: Read CSV

2.3 Basic Operations on the Data

```python
df.head()
```

Listing 3: Show First 5 Rows

Output:

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0	A/5 21171	7.2500	NaN	S
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0	PC 17599	71.2833	C85	C
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0	STON/O2. 3101282	7.9250	NaN	S
3	4	1	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	female	35.0	1	0	113803	53.1000	C123	S
4	5	0	3	Allen, Mr. William Henry	male	35.0	0	0	373450	8.0500	NaN	S

Figure 1: Sample of Titanic Dataset (first 5 rows)

```python
df.info()
```

Listing 4: Data Info

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 891 entries, 0 to 890
Data columns (total 12 columns):
 #   Column       Non-Null Count  Dtype  
---  ------       --------------  -----  
 0   PassengerId  891 non-null    int64  
 1   Survived     891 non-null    int64  
 2   Pclass       891 non-null    int64  
 3   Name         891 non-null    object 
 4   Sex          891 non-null    object 
 5   Age          714 non-null    float64
 6   SibSp        891 non-null    int64  
 7   Parch        891 non-null    int64  
 8   Ticket       891 non-null    object 
 9   Fare         891 non-null    float64
 10  Cabin        204 non-null    object 
 11  Embarked     889 non-null    object 
dtypes: float64(2), int64(5), object(5)
memory usage: 83.7+ KB
```

Figure 2: Data Info

```python
df['Name'].describe()
```

Listing 5: Categorical Data - Name

Output:

```
count                        891
unique                       891
top       Dooley, Mr. Patrick
freq                           1
Name: Name, dtype: object
```

Figure 3: Describe Categorical Data

```python
df['Age'].describe()
```

Listing 6: Numerical Data - Age

Output:

```
count    714.000000
mean      29.699118
std       14.526497
min        0.420000
25%       20.125000
50%       28.000000
75%       38.000000
max       80.000000
Name: Age, dtype: float64
```

Figure 4: Describe Numerical Data

```python
df['Survived'].value_counts(dropna=False)
```

Listing 7: Survival Count

Output:

```
Survived
0    549
1    342
Name: count, dtype: int64
```

Figure 5: Survival Count

2.4 Data Cleaning

```python
print(df.isna().sum())
```

Listing 8: Check NaN Values

Output:

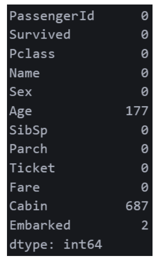


Figure 6: NaN Values Count

```python
meanAge =(int)(df['Age'].median())
print('Mean Age:', meanAge)

df['Age'] = df['Age'].fillna(meanAge)
print('NaN remaining in Age:', df['Age'].isna().sum())
```

Listing 9: Fill NaN in Age with Median

Output:

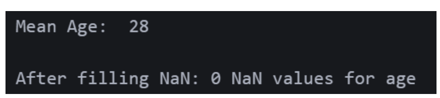


Figure 7: Clean Age Data

```python
df['Cabin'] = df['Cabin'].fillna('Unknown')
print('NaN remaining in Cabin:', df['Cabin'].isna().sum())
```

Listing 10: Fill NaN in Cabin with "Unknown"

Output:

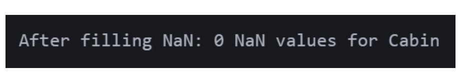
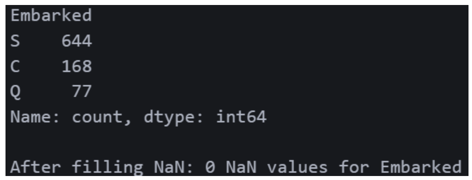


Figure 8: Clean Cabin Data

```python
print(df['Embarked'].value_counts())
df['Embarked'] = df['Embarked'].fillna(df['Embarked'].mode()[0])
print('NaN remaining in Embarked:', df['Embarked'].isna().sum())
```

Listing 11: Fill NaN in Embarked with Mode

Output:



```
Embarked
S    644
C    168
Q     77
Name: count, dtype: int64

After filling NaN: 0 NaN values for Embarked
```

Figure 9: Clean Embarked Data

2.5 Convert Categorical to Numerical

```python
df['Sex'].replace({'female': 0, 'male': 1}, inplace=True)
df['Embarked'].replace({'S': 0, 'C': 1, 'Q': 2}, inplace=True)
```

Listing 12: Encoding Sex and Embarked

2.6 Plottings and Visualizations

```
sns.countplot(data=df, x='Survived', hue='Survived')
plt.title('Survived Bar Plot')
plt.legend(labels=['Died', 'Survived'])
plt.show()
print(df['Survived'].value_counts())
```

Listing 13: Survival Count Plot

Output:

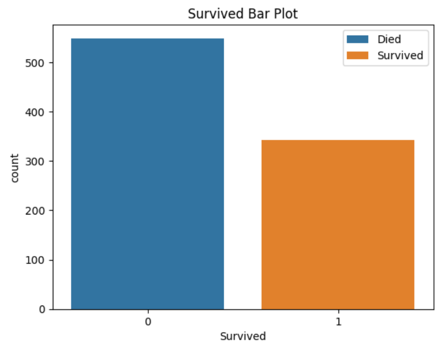


Figure 10

```
sns.countplot(data=df, x='Pclass', hue='Pclass')
plt.title('Passenger Class Distribution')
plt.legend(labels=['First','Second','Third'])
plt.show()
print(df['Pclass'].value_counts())
```

Listing 14: Passenger Class Distribution

Output:

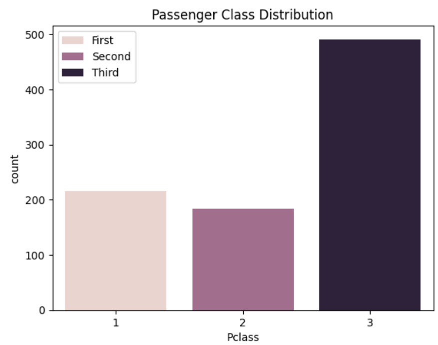


Figure 11

```python
ax = sns.histplot(data=df, x='Age')
for p in ax.patches:
    height = p.get_height()
    if height > 0:
        ax.text(p.get_x() + p.get_width() / 2, height + 0.5,
                int(height), ha='center')
plt.title('Age Distribution')
plt.show()
counts, bin_edges = np.histogram(df['Age'], bins='auto')
print("Counts per bin:", counts)
print("Bin edges:", bin_edges)
```

Listing 15: Age Distribution

Output:

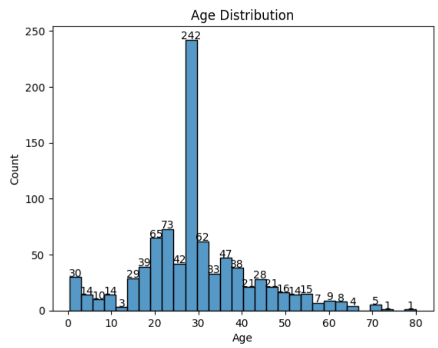


Figure 12

```python
sns.histplot(data=df, x='Fare')
plt.title('Fare Distribution')
plt.show()
```

Listing 16: Fare Distribution

Output:

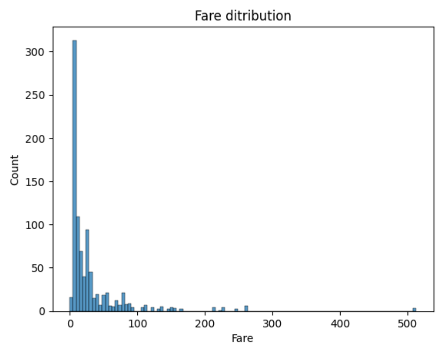


Figure 13

```
sns.histplot(data=df, x='Age', hue='Survived', stat='count', shrink=0.7,
    multiple='dodge')
plt.title('Histogram comparing age for survived')
plt.legend(labels=['Lived', 'Died'])
```

Listing 17: Age Histogram by Survival

Output:

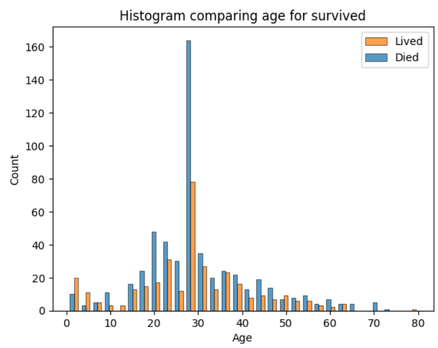


Figure 14

```
d = df.groupby(['Sex', 'Survived']).size().reset_index(name='count')
d['Sex'] = d['Sex'].map({0: 'Female', 1: 'Male'})
d['Survived'] = d['Survived'].map({0: 'Died', 1: 'Survived'})
sns.barplot(data=d, x='Sex', y='count', hue='Survived')
plt.title('Survived grouped by sex')
d
```

Listing 18: Survived Grouped by Sex

Output:

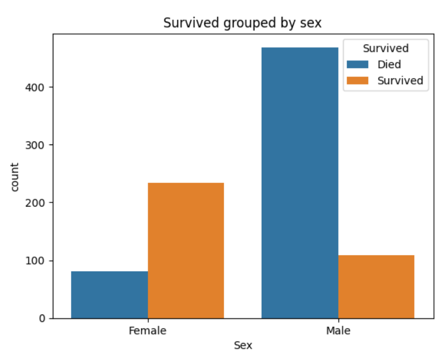


Figure 15

```python
d = df.groupby(['Sex', 'Embarked']).size().reset_index(name='Count')
d['Embarked'] = d['Embarked'].map({1:'Cherbourg', 2:'Queenstown', 0:'Southampton'})
d['Sex'] = d['Sex'].map({0:'Female', 1:'Male'})
sns.barplot(data=d, x='Embarked', y='Count', hue='Sex')
plt.title('Embarked grouped by sex')
d
```

Listing 19: Embarked Grouped by Sex

Output:

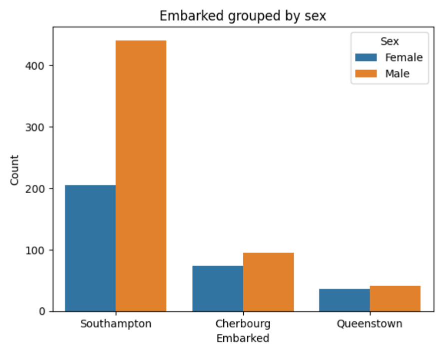


Figure 16

```python
d = df.groupby(['Survived','Sex','Pclass']).size().reset_index(name='Count')
d['Sex'] = d['Sex'].map({0: 'Female', 1:'Male'})
sns.catplot(data=d, x='Sex', y='Count', hue='Pclass', col='Survived', kind='bar')
d
```

Listing 20: Class Distribution by Sex and Survival

Output:

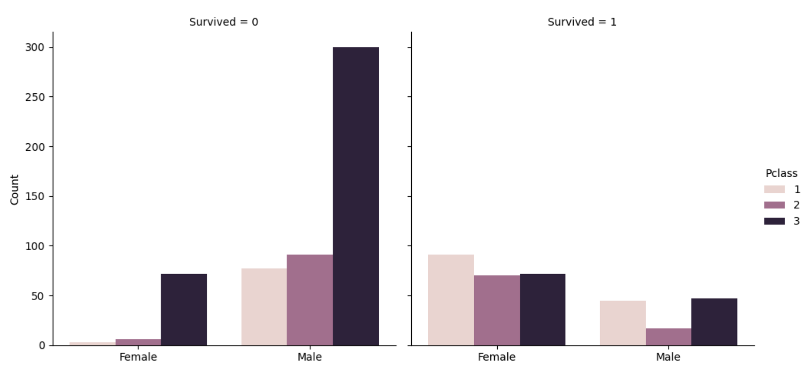


Figure 17

2.7 Scikit-learn

2.7.1 Logistic regression on Titanic Dataset

This section involves using Scikit-learn to train and evaluate classification models for predicting Titanic survival. Logistic Regression is chosen as the primary model in this analysis due to the binary nature of the prediction task: determining whether a passenger survived or not. It is a well-established, interpretable, and efficient algorithm for binary classification problems. Logistic Regression provides probabilistic outputs and clear insights into the contribution of each feature through its coefficients. Additionally, it serves as a strong baseline model in many classification tasks and can be easily enhanced through regularization and hyperparameter tuning. These characteristics make it an ideal starting point for modeling survival prediction using the Titanic dataset.

One-Hot Encoding for Embarked and Pclass

In the preprocessing step, we first reverted the `Embarked` column back to its original categorical string values ('S', 'C', 'Q') because it had been previously mapped to numeric codes (0, 1, 2). This was necessary to correctly apply one-hot encoding, which creates binary indicator variables for each category. Using `pd.get_dummies`, we performed one-hot encoding on both the `Embarked` and `Pclass` columns, converting these categorical features into multiple binary columns. We set `drop_first=True` to avoid multicollinearity by dropping the first category in each feature, and `dtype=int` to ensure the resulting columns are integer type. This transformation allows machine learning models to process categorical variables effectively.

```
df['Embarked'] = df['Embarked'].replace({0: 'S', 1: 'C', 2: 'Q'})  # because already mapped
df = pd.get_dummies(df, columns=['Embarked'], prefix='Embarked', drop_first=True, dtype=int)
df = pd.get_dummies(df, columns=['Pclass'], prefix='Class', drop_first=True, dtype=int)
df
```

Listing 21: One-Hot Encoding of 'Embarked' and 'Pclass'

Output:

	PassengerId	Survived	Name	Sex	Age	SibSp	Parch	Ticket	Fare	Cabin	Embarked_Q	Embarked_S	Class_2	Class_3
0	1	0	Braund, Mr. Owen Harris	1	22.0	1	0	A/5 21171	7.2500	Unknown	0	1	0	1
1	2	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	0	38.0	1	0	PC 17599	71.2833	C85	0	0	0	0
2	3	1	Heikkinen, Miss. Laina	0	26.0	0	0	STON/O2. 3101282	7.9250	Unknown	0	1	0	1
3	4	1	Futrelle, Mrs. Jacques Heath (Lily May Peel)	0	35.0	1	0	113803	53.1000	C123	0	1	0	0
4	5	0	Allen, Mr. William Henry	1	35.0	0	0	373450	8.0500	Unknown	0	1	0	1
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
886	887	0	Montvila, Rev. Juozas	1	27.0	0	0	211536	13.0000	Unknown	0	1	1	0
887	888	1	Graham, Miss. Margaret Edith	0	19.0	0	0	112053	30.0000	B42	0	1	0	0
888	889	0	Johnston, Miss. Catherine Helen "Carrie"	0	28.0	1	2	W./C. 6607	23.4500	Unknown	0	1	0	1
889	890	1	Behr, Mr. Karl Howell	1	26.0	0	0	111369	30.0000	C148	0	0	0	0
890	891	0	Dooley, Mr. Patrick	1	32.0	0	0	370376	7.7500	Unknown	1	0	0	1

891 rows × 14 columns

Figure 18: Encoded Data

Feature Selection and Correlation Heatmap

Before training the machine learning models, we selected a subset of relevant features from the dataset, including both numerical features (such as `Age`, `Fare`, `SibSp`, and `Parch`) and the one-hot encoded categorical features (`Embarked_Q`, `Embarked_S`, `Class_2`, and `Class_3`), as well as the binary encoded `Sex` feature. We then computed the correlation matrix for these features to understand the relationships between them and visualized it using a heatmap with a `coolwarm` color scheme. To standardize the feature scales and improve model performance, we applied `StandardScaler` to normalize the data. Finally, the dataset was split into training and testing sets with a 70%-30% ratio, using stratified sampling to maintain the original class distribution of the target variable `Survived`, ensuring balanced representation in both subsets.

```python
X = df[['Sex', 'Age', 'SibSp', 'Parch', 'Fare', 'Embarked_Q', 'Embarked_S'
    , 'Class_2', 'Class_3']]

correlation = X.corr()
sns.heatmap(correlation, annot=True, cmap='coolwarm', square=True)

scaler = sklearn.preprocessing.StandardScaler()
X_scaled = scaler.fit_transform(X)

y = df['Survived']

X_train, X_test, ytrain, ytest = train_test_split(X_scaled, y, test_size
    =0.3, train_size=0.7,random_state=69, shuffle=True, stratify=y)
```

Listing 22: Selecting and scaling features

Output:

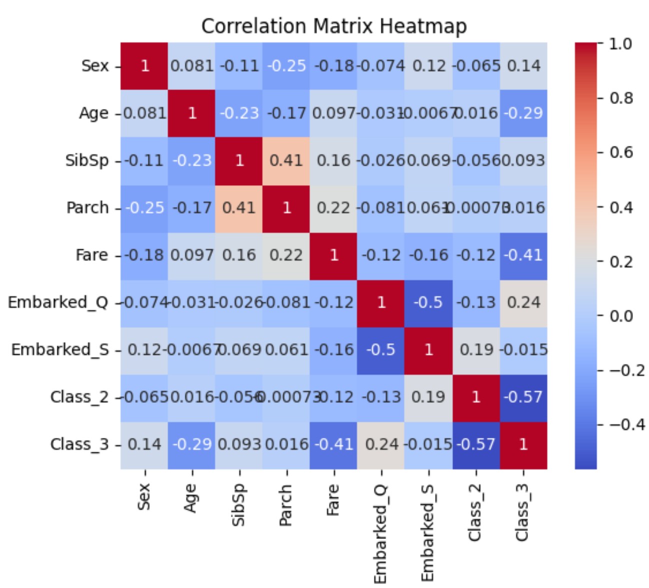


Figure 19

Model Definition and Grid Search

In this section, we use two approaches for logistic regression classification. The first model is a standard Logistic Regression with default parameters, while the second employs `GridSearchCV` to perform hyperparameter tuning over a predefined parameter grid. This allows us to systematically search for the best combination of hyperparameters to improve model performance.

```
parameters = {
    'C': [0.01, 0.1, 1, 10],
    'solver': ['liblinear', 'lbfgs'],
    'penalty': ['l2']
}

model1 = sklearn.linear_model.LogisticRegression()
model2 = sklearn.model_selection.GridSearchCV(LogisticRegression(max_iter
    =100), parameters, cv=5)
```

Listing 23: Define Models and Parameter Grid

Training the Models

```
model1.fit(X_train, ytrain)
model2.fit(X_train, ytrain)
```

Listing 24: Train Both Models

Output:

Table 1 lists the default parameters of the `LogisticRegression` model. The key parameters include:

- `penalty`: Specifies the norm used in the penalization (here L2 regularization).
- `C`: Inverse of regularization strength; smaller values specify stronger regularization.
- `solver`: Algorithm to use in optimization; 'lbfgs' is suitable for small to medium datasets.
- `max_iter`: Maximum number of iterations for the solver to converge.
- Other parameters control aspects like intercept fitting, class weights, and verbosity.

Table 1: LogisticRegression Parameters

Parameter	**Value**
penalty	'l2'
dual	False
tol	0.0001
C	1.0
fit_intercept	True
intercept_scaling	1
class_weight	None
random_state	None
solver	'lbfgs'
max_iter	100
multi_class	'deprecated'
verbose	0
warm_start	False
n_jobs	None
l1_ratio	None

Table 2 shows the parameters used in the `GridSearchCV` process. Important settings include:

- `estimator`: The base logistic regression model to tune.
- `param_grid`: The dictionary specifying the hyperparameters and their possible values to search over.
- `cv`: Number of cross-validation folds to evaluate each parameter combination.
- `refit`: Whether to refit the model with the best found parameters after the search.

Table 2: GridSearchCV Parameters

Parameter	**Value**
estimator	LogisticRegression()
param_grid	{'C': [0.01, 0.1, ...], 'penalty': ['l2'], 'solver': ['liblinear', 'lbfgs']}
scoring	None
n_jobs	None
refit	True
cv	5
verbose	0
pre_dispatch	'2*n_jobs'
error_score	nan
return_train_score	False
best_estimator_	LogisticRegression

Table 3 summarizes the best hyperparameters found by `GridSearchCV` after evaluating all combinations. Notably, the value of `C` was optimized to 0.01, indicating stronger regularization than the default. The solver and penalty remained the same, ensuring a balance between model complexity and performance.

Table 3: Best LogisticRegression Parameters

Parameter	**Value**
penalty	‘l2’
dual	False
tol	0.0001
C	0.01
fit_intercept	True
intercept_scaling	1
class_weight	None
random_state	None
solver	‘lbfgs’
max_iter	100
multi_class	‘deprecated’
verbose	0
warm_start	False
n_jobs	None
l1_ratio	None

Training Accuracy

Before evaluating the models on the test data, we assess how well each model performs on the training data to understand their learning behavior. Model 1 (basic logistic regression) achieved a training accuracy of 79%, while Model 2 (logistic regression with GridSearchCV optimization) slightly improved the performance with an accuracy of 80%. These results indicate that both models fit the training data reasonably well without severe overfitting.

```
trainPred1 = model1.predict(X_train)
trainAcc1 = sklearn.metrics.accuracy_score(ytrain, trainPred1)

trainPred2 = model2.predict(X_train)
trainAcc2 = sklearn.metrics.accuracy_score(ytrain, trainPred2)
```

Listing 25: Train Accuracy Scores

Output:

```
Train accuracy model 1:  0.7961476725521669
Train accuracy model 2:  0.8025682182985554
```

Figure 20

Testing and Evaluation

After training the models, we evaluated their performance on the test set to assess their generalization capability. Model 1 achieved an accuracy of 81%, while Model 2 slightly outperformed it with an accuracy of 82%. This improvement suggests that the hyperparameter tuning performed by GridSearchCV in Model 2 helped enhance its ability to generalize better to unseen data.

```python
ypred1 = model1.predict(X_test)
testAcc1 = sklearn.metrics.accuracy_score(ytest, ypred1)

ypred2 = model2.predict(X_test)
testAcc2 = sklearn.metrics.accuracy_score(ytest, ypred2)
```

Listing 26: Test Accuracy Scores

Output:

```
Test accuracy model 2:  0.8134328358208955
Test accuracy model 2:  0.8246268656716418
```

Figure 21

Confusion Matrix and Classification Report

To gain deeper insight into the performance of each model beyond overall accuracy, we used confusion matrices and classification reports. The confusion matrices show how well each model distinguishes between passengers who survived and those who did not, highlighting the number of true positives, true negatives, false positives, and false negatives. The classification reports provide additional metrics including precision, recall, and F1-score for both classes (Survived and Died), offering a more detailed evaluation of each model's strengths and weaknesses in handling imbalanced or overlapping classes.

```python
cm1 = confusion_matrix(ytest, ypred1)
sns.heatmap(cm1, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Died', 'Survived'], yticklabels=['Died', 'Survived'])
plt.title('Confusion Matrix model 1')
plt.show()

cm2 = confusion_matrix(ytest, ypred2)
sns.heatmap(cm2, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Died', 'Survived'], yticklabels=['Died', 'Survived'])
plt.title('Confusion Matrix model 2')
plt.show()

print('Model 1 classification report:\n',
      sklearn.metrics.classification_report(y_true=ytest, y_pred=ypred1))
print('Model 2 classification report:\n',
```

```
        sklearn.metrics.classification_report(y_true=ytest, y_pred=ypred2))
```
Listing 27: Confusion Matrix and Classification Report

Output:

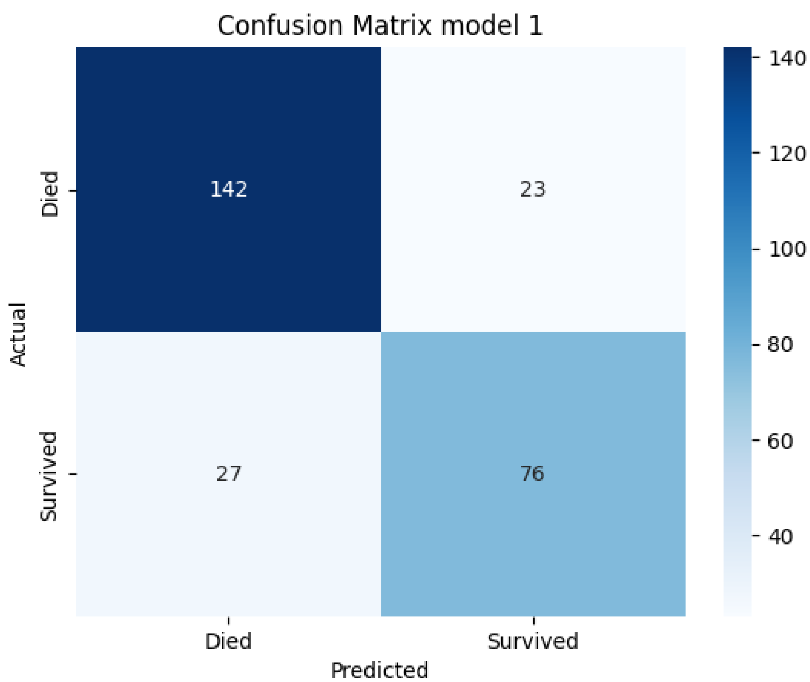


Figure 22

Table 4: Classification Report for Model 1

Class/Avg	**Precision**	**Recall**	**F1-score**	**Support**
0 (Died)	0.84	0.86	0.85	165
1 (Survived)	0.77	0.74	0.75	103
Accuracy		0.81		268
Macro avg	0.80	0.80	0.80	268
Weighted avg	0.81	0.81	0.81	268

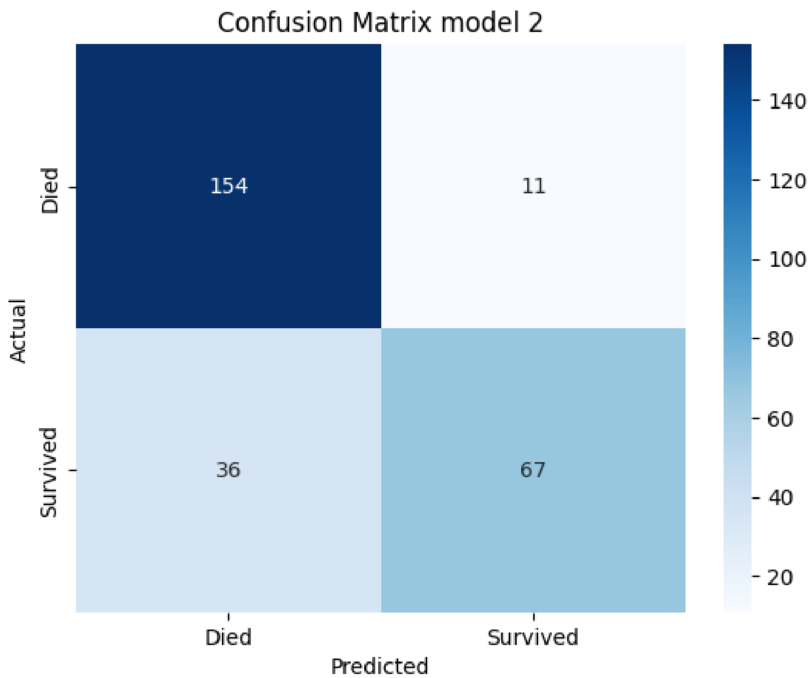


Figure 23

Table 5: Classification Report for Model 2

Class/Avg	Precision	Recall	F1-score	Support
0 (Died)	0.81	0.93	0.87	165
1 (Survived)	0.86	0.65	0.74	103
Accuracy		0.82		268
Macro avg	0.83	0.79	0.80	268
Weighted avg	0.83	0.82	0.82	268

The classification reports summarize the performance of the two models. Model 1, the basic Logistic Regression without hyperparameter tuning, achieved an accuracy of 81%, with balanced precision and recall scores for both classes. Model 2, which used GridSearchCV for hyperparameter optimization, slightly improved the overall accuracy to 82%. While Model 2 achieved a higher precision (0.86) in predicting class 1 (Survived), it had a lower recall (0.65), indicating more false negatives compared to Model 1. On the other hand, Model 2 performed better in identifying class 0 (Died), with a recall of 0.93 compared to 0.86 in Model 1. Overall, the optimized model showed marginal improvement in accuracy and a shift in the balance between precision and recall, which might be more favorable depending on whether false positives or false negatives are more critical in the context of prediction.

2.7.2 K-Nearest Neighbors on Iris Dataset

To extend our classification analysis to a multi-class setting, we used the Iris dataset, which includes three species of iris flowers: *Iris-setosa*, *Iris-versicolor*, and *Iris-virginica*. Unlike the Titanic dataset which is a binary classification problem, this dataset allows us to evaluate K-Nearest Neighbors (KNN) in a more general classification context.

Data Preparation

We first read the dataset and encoded the target variable (species) numerically using:

```python
df['Species'] = df['Species'].replace({'Iris-setosa':0, 'Iris-versicolor'
    :1, 'Iris-virginica':2})
```

Listing 28: Encoding Species Labels

Then, we split the data into features and target labels:

```python
features = df.drop('Species', axis=1)
target = df['Species']
```

Listing 29: Splitting Features and Target

Model Training with KNN

We used an 80/20 train-test split, ensuring class stratification:

```python
X_train, X_test, ytrain, ytest = train_test_split(features, target,
     train_size=0.8, test_size=0.2, random_state=49, shuffle=True, stratify
    =target)

model = KNeighborsClassifier(n_neighbors=4)
model.fit(X_train, ytrain)
```

Listing 30: Train-Test Split and Model Creation

Model Evaluation

We evaluated both training and testing accuracy:

```python
trainAcc = accuracy_score(ytrain, model.predict(X_train))
testAcc = accuracy_score(ytest, model.predict(X_test))
```

Listing 31: Evaluation

Training Accuracy: 100%
Testing Accuracy: 100%

We also plotted the confusion matrix to visualize classification results across the three flower classes:

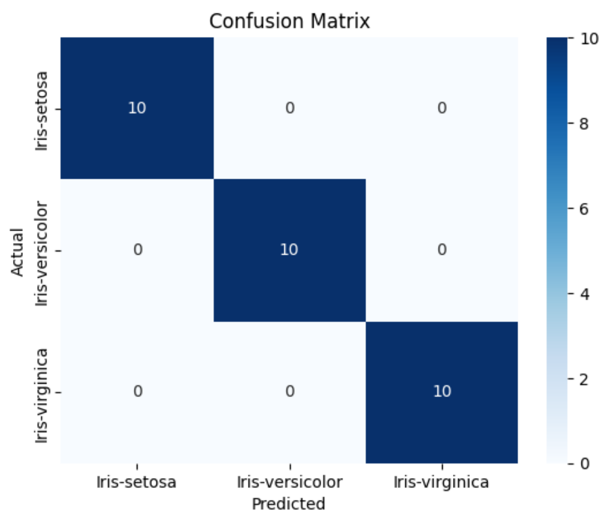


Figure 24: Confusion Matrix for Iris KNN Model

Finally, we generated a classification report summarizing precision, recall, and F1-score for each class:

```python
print(classification_report(ytest, ypred))
```

Listing 32: Classification Report

Classification Report Output:

Table 6: Classification Report for KNN on Iris Dataset

Class/Avg	**Precision**	**Recall**	**F1-score**	**Support**
0 (Iris-setosa)	1.00	1.00	1.00	10
1 (Iris-versicolor)	1.00	1.00	1.00	10
2 (Iris-virginica)	1.00	1.00	1.00	10
Accuracy		1.00		30
Macro avg	1.00	1.00	1.00	30
Weighted avg	1.00	1.00	1.00	30

This experiment demonstrates that KNN is well-suited for multi-class problems like the Iris dataset, especially when the dataset is balanced and of small to moderate size.

3 Conclusion

In this report, we explored two supervised machine learning problems using the Titanic and Iris datasets. For the Titanic dataset, we performed binary classification using logistic regression to predict passenger survival based on various features. For the Iris dataset, we applied the K-Nearest Neighbors (KNN) algorithm to solve a multiclass classification problem. Throughout both case studies, we carried out important preprocessing tasks (handling missing data, encoding categorical features, scaling, and feature selection), evaluated model performance using accuracy scores and confusion matrices, and interpreted results through classification reports.

This hands-on experience not only demonstrated the technical steps of building and evaluating models, but also highlighted how real-world data can be structured and used to generate intelligent predictions. These insights are directly transferable to the kind of data-driven personalization and intelligence 3VO could benefit from across its digital ecosystem—including the mobile/web app and the Telegram Mini App.

As users interact with 3VO's platforms—through content views, likes, messages, shares, engagement time, or agent interactions—the system can collect a rich stream of behavioral and contextual data. Just like in our projects where features led to meaningful predictions, 3VO can leverage similar data pipelines to build recommendation systems (e.g., suggesting creators, AI agents, or relevant posts), user segmentation models (e.g., identifying community leaders, passive users, or high-engagement profiles), and predictive systems (e.g., forecasting drop-off likelihood or preferred interaction types).

Ultimately, by applying machine learning models to user data, 3VO can dynamically adapt to each user, improve content discovery, optimize engagement strategies, and build an AI-enhanced experience that evolves with the community in real-time. This approach turns raw interaction data into actionable insights—just as we transformed simple CSV files into predictive models throughout this report.