Task Report – Titanic Survival Prediction

Introduction

The Titanic disaster, which occurred on April 15, 1912, is one of the most tragic events in maritime history. This project focuses on building a machine-learning model that predicts whether a passenger survived the Titanic disaster based on the available data. Using the Kaggle dataset, we applied several data preprocessing steps, feature engineering techniques, and machine learning algorithms to achieve accurate predictions. This report summarises the process and approach taken to solve the problem.

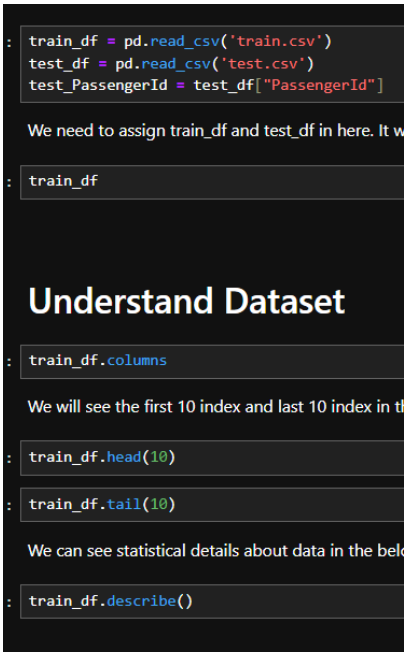
Problem Statement

The task is to predict the survival of Titanic passengers based on a dataset containing passenger details. The training set contains the known outcome (Survived), while the test set lacks this information. The objective is to build a model that accurately predicts the survival of passengers in the test set.

Solution Approach

1. Data Exploration and Visualization

We begin by loading the dataset and exploring its structure to understand the features and identify any issues such as missing values or outliers.



This code loads the training and test datasets and provides summary statistics. The describe() function reveals statistical details of numerical features, and info() identifies missing values and data types.

Next, we visualize the distribution of key categorical features such as Survived, Pclass, Sex, and Embarked using bar plots.

```python
def bar_plot(variable):
    """
        input: variable ex: "Sex"
        output: bar plot & value count
    """
    # get feature
    var = train_df[variable]
    # count number of categorical variable(value/sample)
    varValue = var.value_counts()

    # visualize
    plt.figure(figsize = (12,4))
    plt.bar(varValue.index, varValue)
    plt.xticks(varValue.index, varValue.index.values)
    plt.ylabel("Frequency")
    plt.title(variable)
    plt.show()
    print("{}: \n {}".format(variable,varValue))
```

This function generates bar plots for categorical variables. For example, it shows that more passengers from lower classes (Pclass 3) did not survive, and female passengers had a higher survival rate compared to males.

2. Handling Missing Data

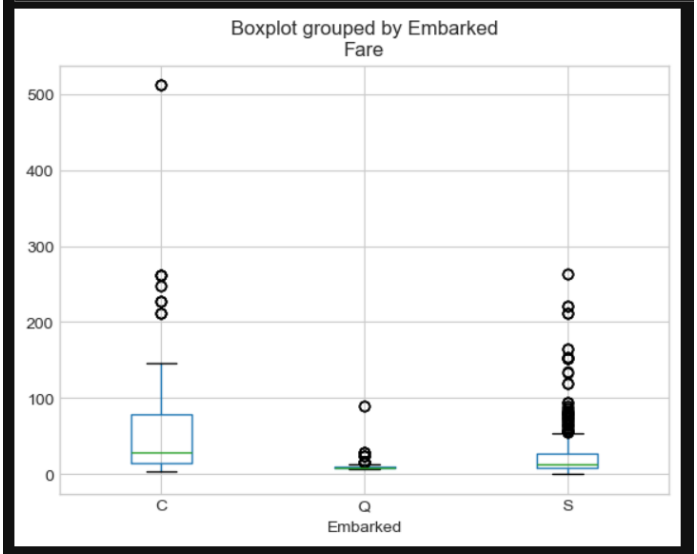
Missing data can affect model performance, so we first visualize and calculate the percentage of missing values in the dataset.

```python
train_df.isnull().sum()
```

```
PassengerId       0
Survived        418
Pclass            0
Name              0
Sex               0
Age             256
SibSp             0
Parch             0
Ticket            0
Fare              1
Cabin          1007
Embarked          2
dtype: int64
```

The above snippet shows where missing values occur, with features like Cabin having many missing values. We decided to drop some features with too many missing values (Cabin, Ticket, etc.), while for other columns, we impute missing values.

```
train_df.boxplot(column="Fare",by = "Embarked")
plt.show()
```



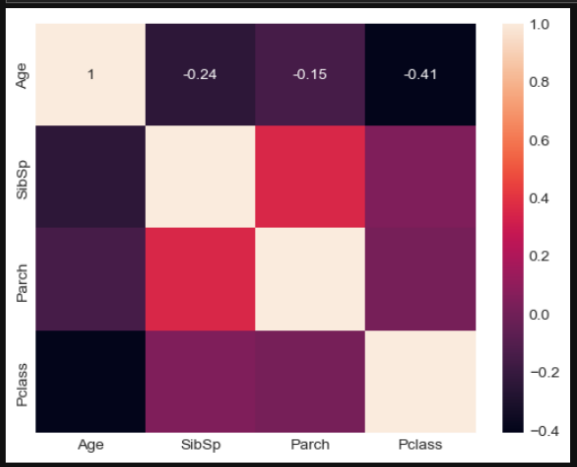
We can fill according to boxplot table.

```
train_df["Embarked"] = train_df["Embarked"].fillna("C")
train_df[train_df["Embarked"].isnull()]
```

We filled missing Embarked values with 'C' (Cherbourg) after checking the distribution of the Fare feature. The median fare for missing values corresponded with passengers who likely embarked from Cherbourg.

For missing Age values, we applied a more nuanced approach by using the median age of passengers with similar family and class characteristics.

```
sns.heatmap(train_df[["Age","SibSp","Parch","Pclass"]].corr(), annot = True)
plt.show()
```



We will fill according to correlation relationship. Considering the correlation with these features, we can follow a method like the one below to fill in the missing values!

```
index_nan_age = list(train_df["Age"][train_df["Age"].isnull()].index)
for i in index_nan_age:
    age_pred = train_df["Age"][((train_df["SibSp"] == train_df.iloc[i]["SibSp"]) &(train_df["Parch"] == train_df.iloc[i]["Parch"])& (train_df["Pclass"] =
    age_med = train_df["Age"].median()
    if not np.isnan(age_pred):
        train_df["Age"].iloc[i] = age_pred
    else:
        train_df["Age"].iloc[i] = age_med
```

This strategy fills missing age values by using the median age of passengers with similar family structure (SibSp, Parch) and class (Pclass). This approach improves the quality of imputed data and preserves relationships between features.

3. Outlier Detection and Removal

Outliers can distort the performance of machine learning models, so we identified and removed them using the **Interquartile Range (IQR)** method.

```
def detect_outliers(df,features):
    outlier_indices = []

    for c in features:
        # 1st quartile
        Q1 = np.percentile(df[c],25)
        # 3rd quartile
        Q3 = np.percentile(df[c],75)
        # IQR
        IQR = Q3 - Q1
        # Outlier step
        outlier_step = IQR * 1.5
        # detect outlier and their indeces
        outlier_list_col = df[(df[c] < Q1 - outlier_step) | (df[c] > Q3 + outlier_step)].index
        # store indeces
        outlier_indices.extend(outlier_list_col)

    outlier_indices = Counter(outlier_indices)
    multiple_outliers = list(i for i, v in outlier_indices.items() if v > 2)

    return multiple_outliers
```

```
train_df.loc[detect_outliers(train_df,["Age","SibSp","Parch","Fare"])]
```

Outliers were detected and removed based on numerical features such as Age, Fare, SibSp, and Parch. This helps improve model accuracy by removing passengers whose values were significantly different from the rest of the data.

4. Feature Engineering

Title Extraction: We extracted titles from the Name feature to create a new feature, Title, which captures useful information about passengers' social status.

```
train_df['Title'] = train_df['Name'].str.split(', ', expand=True)[1].str.split('.', expand=True)[0]
train_df['Is_Married'] = 0
train_df['Is_Married'].loc[train_df['Title'] == 'Mrs'] = 1
```

```
train_df['Title'] = train_df['Title'].replace(['Miss', 'Mrs','Ms', 'Mlle', 'Lady', 'Mme', 'the Countess', 'Dona'], 'Miss/Mrs/Ms')
train_df['Title'] = train_df['Title'].replace(['Dr', 'Col', 'Major', 'Jonkheer', 'Capt', 'Sir', 'Don', 'Rev'], 'Dr/Military/Noble/Clergy')

sns.barplot(x=train_df['Title'].value_counts().index, y=train_df['Title'].value_counts().values, ax=axs[1])
axs[1].set_title('Title Feature Value Counts After Grouping', size=20, y=1.05)

plt.show()
```

Titles were grouped to reduce complexity, and similar titles were consolidated. This improves the model's understanding of gender and social status, which impacts survival rates.

Family Size and Grouping: We created a Family_Size feature by adding SibSp and Parch, and then grouped it into categories like Alone, Small, and Large.

```
train_df['Family_Size'] = train_df['SibSp'] + train_df['Parch'] + 1
family_map = {1: 'Alone', 2: 'Small', 3: 'Small', 4: 'Small', 5: 'Medium', 6: 'Medium', 7: 'Large', 8: 'Large', 11: 'Large'}
train_df['Family_Size_Grouped'] = train_df['Family_Size'].map(family_map)
train_df.head()
```

Family size can influence survival since passengers traveling alone or in small families may have different survival rates compared to large families.

5. Data Encoding

We used **Label Encoding** and **One-Hot Encoding** to convert categorical features into a numerical format so that machine learning algorithms can process them.

```
non_numeric_features = ['Embarked', 'Sex', 'Title', 'Family_Size_Grouped','Age', 'Fare']

label_encoder = LabelEncoder()

for column in non_numeric_features:
    train_df[column] = label_encoder.fit_transform(train_df[column])
```

```
cat_features = ['Pclass', 'Sex', 'Embarked', 'Title', 'Family_Size_Grouped']
one_hot_encoder = OneHotEncoder()
encoded_features = one_hot_encoder.fit_transform(train_df[cat_features]).toarray()

column_names = []
for i, column in enumerate(cat_features):
    unique_labels = train_df[column].unique()

    names = [f"{column}_{label}" for label in unique_labels]
    column_names.extend(names)

one_hot_encoded_df = pd.DataFrame(encoded_features, columns=column_names)

train_df = pd.concat([train_df, one_hot_encoded_df], axis=1)
```

Label Encoding converts categorical variables to integers, while **One-Hot Encoding** creates binary columns for non-ordinal categorical variables like Pclass, Sex, and Embarked. This helps ensure the model correctly interprets categorical data.

6. Model Training and Evaluation

We split the training data into training and testing sets and trained several classification models, including **Logistic Regression**, **Decision Tree**, **Random Forest**, **SVM**, and **XGBoost**. We compared model performance based on accuracy.

```python
models = {
    'Logistic Regression': LogisticRegression(),
    'Decision Tree': DecisionTreeClassifier(),
    'Random Forest': RandomForestClassifier(),
    'SVM': SVC(),
    'KNN': KNeighborsClassifier(),
    'XGBoost': xgb.XGBClassifier()
}

for model_name, model in models.items():
    # Train the model
    model.fit(X_train, y_train)

    # Predict on the test set
    y_pred = model.predict(X_test)

    # Evaluate accuracy
    accuracy = accuracy_score(y_test, y_pred)

    print(f'{model_name}: Accuracy = {accuracy:.4f}')
```

```
Logistic Regression: Accuracy = 0.8351
Decision Tree: Accuracy = 0.7938
Random Forest: Accuracy = 0.8076
SVM: Accuracy = 0.6529
KNN: Accuracy = 0.6667
XGBoost: Accuracy = 0.8144
```

This code trains each model and evaluates its performance using **accuracy** as the metric. The best-performing model was **Logistic regression**, followed closely by **XGBoost**. Random Forest models are effective in capturing complex patterns in the data.

7. Generating Final Predictions

After selecting **Logistic Regression** as the final model, we used it to predict the survival of passengers in the test dataset.

```python
logistic_regression = LogisticRegression()
logistic_regression = logistic_regression.fit(X_train, y_train)
print(accuracy_score(logistic_regression.predict(X_test),y_test))
```

```
0.8350515463917526
```

Prediction and Submit

We will predict and submit our .csv file. We can finish this notebook in here!

```python
test_survived = pd.Series(logistic_regression.predict(test), name = "Survived").astype(int)
results = pd.concat([test_PassengerId, test_survived],axis = 1)
results.to_csv("submission.csv",header=True, index = False)
```

We generated the final predictions and saved them in the required CSV format for submission to Kaggle.

Important Resources

Throughout the development of this project, I relied on several resources to gain insights and improve my solution. These resources helped me understand best practices for handling missing data, feature engineering, and regression modelling:

i. AI/ML Roadmap (ai-ml-roadmap.vercel.app)
ii. What is Exploratory Data Analysis? | by Prasad Patil | Towards Data Science
iii. 11 Data Visualization Techniques for Every Use-Case with Examples | DataCamp
iv. Interquartile Range to Detect Outliers in Data - GeeksforGeeks
v. Box Plot - GeeksforGeeks
vi. ML | Handling Missing Values - GeeksforGeeks
vii. Complete Machine Learning In 6 Hours| Krish Naik - YouTube