CS 5785 Homework 1

Cornell Tech, Fall 2023

Yufan Zhang
yz2894@cornell.edu

September 2023

1 Programming Exercises

1.1 The Housing Prices

1.1.1 Examples of continuous and categorical variables

After developing the Python code in Listing 1 to detect the data types of each feature:

Listing 1: Detecting the data types of each feature

```
column_types = df.dtypes

num_features = column_types[column_types != "object"].index.tolist()
print(f"Numerical features: {num_features}")
cat_features = column_types[column_types == "object"].index.tolist()
print(f"Categorical features: {cat_features}")
```

Three examples of numerical (continuous) features include:

```
"LotFrontage", "LotArea", "MasVnrArea"
```

Three examples of categorical features include:

```
"HouseStyle", "ExterQual", "BsmtCond"
```

Let `LotArea` from the numerical features and `HouseStyle` from the categorical features to be the two features that to illustrate the distribution. Figure 1 shows the distribution of these two features. The code in Listing 2 is used to plot the histogram to illustrate the distribution of `LotArea` and `HouseStyle`.

Listing 2: Plotting the histogram to illustrate the distribution of `LotArea` and `HouseStyle`

```
# plot the histogram to illustrate the distribution of "LotArea"
plt.figure(figsize=(10, 6))
sns.distplot(df["LotArea"], bins=50, kde=False)
plt.title("Distribution of LotArea")
plt.xlabel("LotArea")
```

```
plt.ylabel("Count")
plt.savefig(os.path.join(IMG_PATH, "LotArea.png"))

# plot the histogram to illustrate the distribution of "HouseStyle"
plt.figure(figsize=(10, 6))
sns.countplot(x=df["HouseStyle"])
plt.title("Distribution of HouseStyle")
plt.xlabel("HouseStyle")
plt.ylabel("Count")
plt.savefig(os.path.join(IMG_PATH, "HouseStyle.png"))
```

1.1.2 Data preprocessing steps

Feature selection:

(This step has been conducted by discussion with Tian Jin from the same class.) Based on the data description, we can identify the features that best represent the relationship. By doing so, we can filter out the features that do not make big impact on people's decision on house purchase. For example, the `Street` feature is removed as it has only two values to represent the type of road access to the property, which is not likely to influence the decision. The selected features are shown in Listing 3.

Listing 3: Features selected with human knowledge

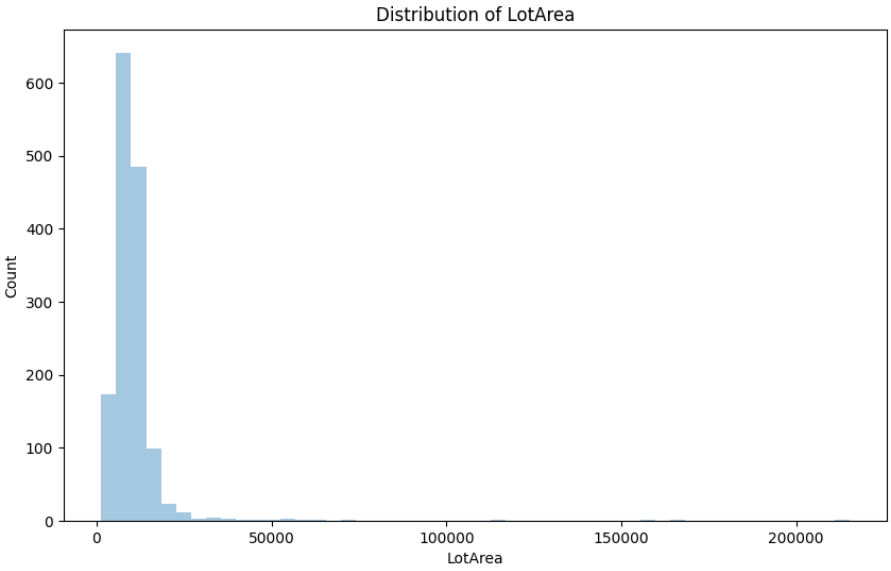
```
num_features_to_keep = [
    "LotFrontage", "LotArea", "OverallQual",
    "OverallCond", "YearBuilt", "YearRemodAdd",
    "MasVnrArea", "TotalBsmtSF", "GrLivArea",
    "BsmtFullBath", "BsmtHalfBath", "FullBath",
    "HalfBath", "BedroomAbvGr", "KitchenAbvGr",
    "TotRmsAbvGrd", "Fireplaces", "GarageArea",
    "PoolArea", "YrSold",
]

cat_features_to_keep = [
    "MSZoning", "Utilities", "LandSlope",
    "BldgType", "HouseStyle", "ExterQual",
    "ExterCond", "BsmtQual", "BsmtCond",
    "HeatingQC", "CentralAir", "KitchenQual",
    "FireplaceQu", "GarageQual", "GarageCond",
    "PoolQC", "SaleType",
]
```

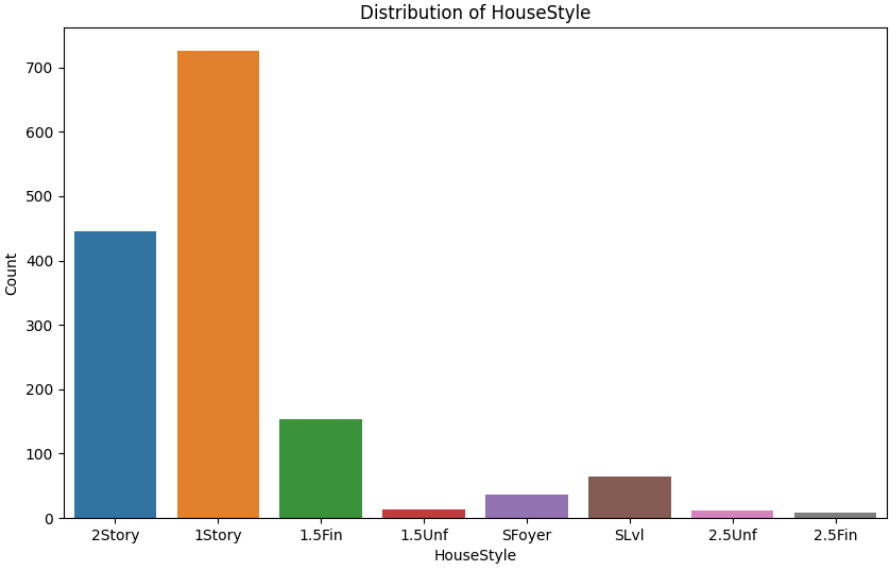
Handling missing values:

Handling missing values is a critical step in the data preprocessing, ensuring data completeness and accuracy; my approach involves dropping features with over 50% missing values, and for categorical attributes, I will impute missing values using the mode, while for numerical features, I will employ the mean as the imputation strategy.

The code in Listing 4 is drop features with high amount of missing values. The features with over 50% missing values are dropped since they are not likely to provide useful information. Those features include `FireplaceQu`, `PoolQC`.



(a) Distribution of `LotArea`



(b) Distribution of `HouseStyle`

Figure 1: Distribution of `LotArea` and `HouseStyle` features

Listing 4: Drop the features with high amount of missing values

```
def get_feature_to_drop_by_hight_na(df_train, feature_list, threshold):
    null_counts = df_train.loc[:, feature_list].isna().sum() / len(df_train)
    return list(null_counts[null_counts > threshold].index)

features_to_drop_by_hight_na = get_feature_to_drop_by_hight_na(
    df, df.columns, threshold=0.5)
print(f"Features to drop: {features_to_drop_by_hight_na}")
df = df.drop(features_to_drop_by_hight_na, axis=1)
```

One thing to be noted is that the missing value should be filled with the mode/mean of that feature in the training data, instead of the whole dataset. This is because the test data should not be used to influence the training process. The code in Listing 5 is used to impute the missing values in the dataset.

Listing 5: Fill the missing values with the measures of the feature in the training data

```
# Get the mathematical measures of each feature
def get_fea_measures_dict(df_train, feature_list, measure="mode"):
    measure_dict = dict()
    for col in feature_list:
        if measure == "mode":
            measure_dict[col] = df_train[col].mode()[0]
        elif measure == "mean":
            measure_dict[col] = df_train[col].mean()
        elif measure == "median":
            measure_dict[col] = df_train[col].median()
        else:
            raise ValueError("measure should be mode/mean/median")
    return measure_dict

cat_mode_dict = get_fea_measures_dict(df, cat_features, "mode")
num_mean_dict = get_fea_measures_dict(df, num_features, "mean")

# Fill the NA values in categorical/numerical features with the mode/mean of the feature
def fill_na_with_measures(df, measure_dict):
    """
    Fill the NA values in features with the measures of the features in the training dat

    Args:
        df: a Pandas dataframe containing the data to be transformed
        measure_dict: a dict containing the measure of each feature

    Return:
        A Pandas dataframe after NA values being filled
    """
    for col in measure_dict.keys():
        df[col].fillna(measure_dict[col], inplace=True)
    return df

df = fill_na_with_measures(df, cat_mode_dict)
```

```
df = fill_na_with_measures(df, num_mean_dict)
```

Transforming ordinal features:

Transforming ordinal features, originally categorical, into numerical representations allows us to capture and utilize the inherent order and hierarchy within these features for more meaningful analysis and modeling, enhancing our understanding of the data's relationships. Consequently, there are several ordinal features within the categorical variables, and our next step involves converting them into numerical representations. For example, all the features that contain the word "Qual" are ordinal features, as they measures the quality of certain aspects of the house.

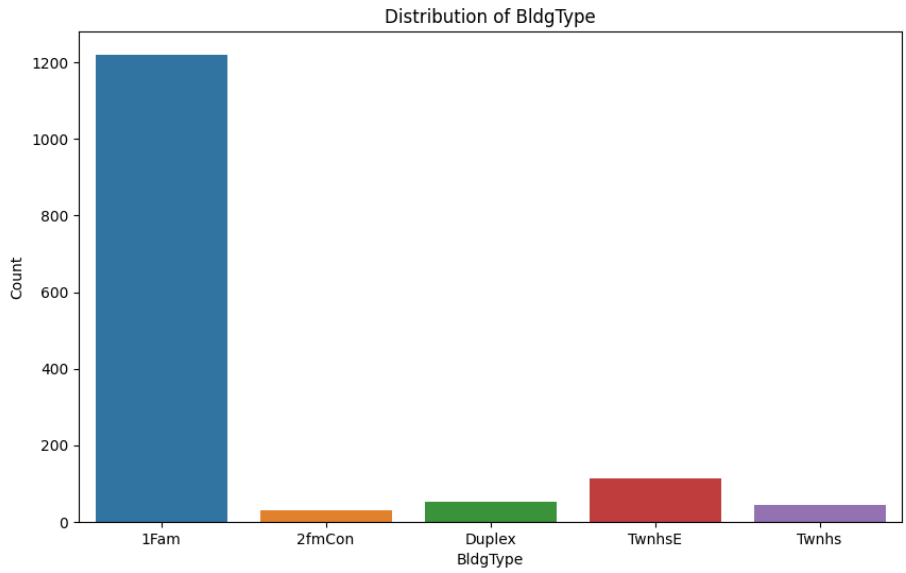


Figure 2: Distribution of `BldgType`

Dropping highly-skewed features:

Dropping highly-skewed features is essential because they do not provide valuable information and can potentially introduce noise into our analysis; as a result, we will remove these skewed features in the following steps. The code in Listing 6 is used to drop the features with high skewness. It turned out that most of the categorical features remaining in the dataset are highly skewed, so I decided to drop all of them except for `HouseStyle`. For example, Figure 2 shows the skewness of the `BldgType` feature.

Listing 6: Drop the features with high skewness

```
def get_skewed_features(df, feature_list, threshold=0.7):
    skewed_features = list()
    for col in feature_list:
        if df[col].value_counts(normalize=True).values[0] > threshold:
            skewed_features.append(col)
    return skewed_features

feature_to_drop_skewed = get_skewed_features(df, cat_features)
print(f"Features to drop: {feature_to_drop_skewed}")
df = df.drop(feature_to_drop_skewed, axis=1)
```

One-hot encoding:

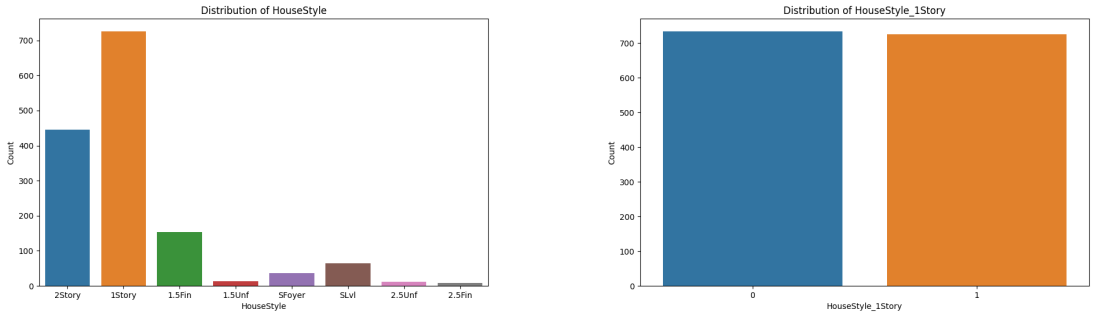
One-hot encoding is a common technique used to transform categorical features into numerical representations. It is a process of converting categorical features into a vector representation that can be easily used by machine learning algorithms. The code in Listing 7 is used to one-hot encode the categorical features.

Listing 7: One-hot encode the categorical features

```
# Create a dict containing all the feature names of each categorical feature
def get_categories_list(df_train, categorical_features):
    categories = dict()
    for col in categorical_features:
        categories[col] = df_train[col].unique().tolist()
    return categories

# Perform one-hot encoding to the training dataset
def apply_one_hot_encode(df, categories_dict):
    for col, categories in categories_dict.items():
        for category in categories:
            df[f"{col}_{category}"] = (df[col] == category).astype(int)
        df.drop(col, axis=1, inplace=True)
    return df

categories_dict = get_categories_list(df, cat_features)
df = apply_one_hot_encode(df, categories_dict)
```



(a) Before one-hot encoding

(b) After one-hot encoding

Figure 3: Distribution of `HouseStyle` before and after one-hot encoding

In the remaining features in the dataset, `HouseStyle` is the only categorical feature that is applicable to one-hot encoding. By performing one-hot encoding, we can not only transform the categorical feature into numerical representations, but also avoid introducing the order of the categories of `HouseStyle` into the model. Figure 3 shows the distribution of the `HouseStyle` feature before and after one-hot encoding. It can be seen that the distribution of the feature is not skewed after one-hot encoding.

Feature scaling:

Feature scaling is necessary to ensure that all the features in a dataset have similar scales or ranges, as many machine learning algorithms are sensitive to the magnitude of feature values. The code in Listing 8 is used to scale the numerical features in the dataset.

Listing 8: Scale the numerical features in the dataset

```
from sklearn.preprocessing import Normalizer
X_normalized = Normalizer().fit_transform(df)
```

1.1.3 Ordinary least squares (OLS): training and testing

The code in Listing 9 is used to implement the OLS model without Scikit-learn:

Listing 9: OLS implementation

```
# Compute OLS from scratch
def OLS(X, Y):
    theta = np.linalg.inv(X.T.dot(X)).dot(X.T).dot(Y)
    return theta

# Fit the model
theta = OLS(X_normalized, target)

# Inference
pred = X_normalized.dot(theta)
```

The mean squared error (MSE) of the OLS model is 974304111.65, and the R^2 score is 0.69103. After training the model, the test set is loaded and preprocessed in the same way as the training set. Then, the model is used to predict the survival of passengers in the test set. The predictions are saved in a CSV file and submitted to Kaggle. The public score on the test set is 1.24275, which is ranked 3,890 out of 3.985 teams on the leaderboard as in Figure 4.

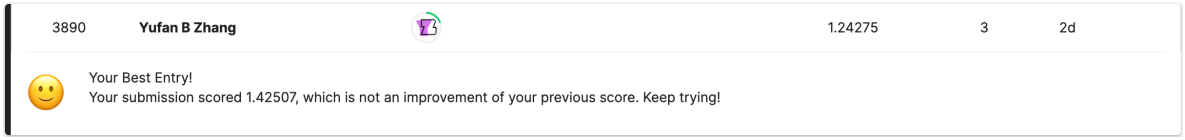


Figure 4: My Submission on Kaggle leaderboard

1.2 The Titanic Disaster

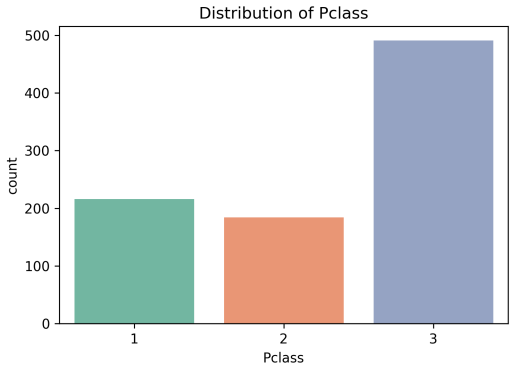
1.2.1 Data preprocessing steps

First, two features that do not provide any useful information are removed from the dataset: `Ticket` and `Name`. Then, the feature with more than 50% missing values, `Cabin`, is also removed. By doing so, I can reduce the feature space to 9 features:

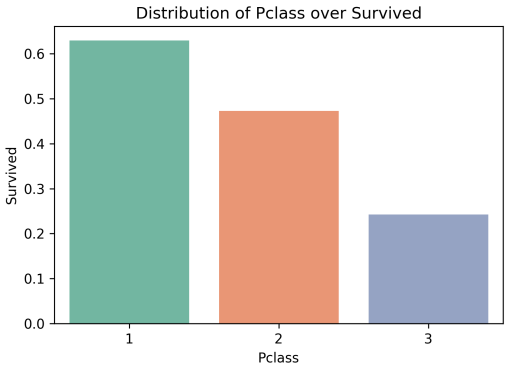
```
['Pclass', 'Age', 'SibSp', 'Parch', 'Fare', 'Sex', 'Embarked']
```

Next, each remaining feature is examined to see if its own distribution, its correlation with the target variable, or its correlation with other features can provide any useful information. This step is highly motivated by Titanic Data Science Solutions on Kaggle.

Pclass: The distribution of `Pclass` is shown in Figure 5. It can be seen that the majority of the passengers are in class 3, followed by class 1 and class 2. Also, the survival rate of passengers in class 1 is higher than that of passengers in class 2, which is higher than that of passengers in class



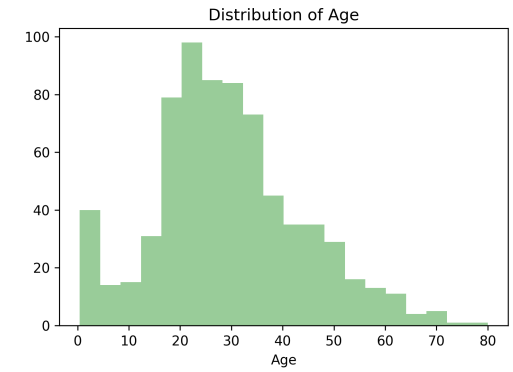
(a) Distribution of `Pclass`



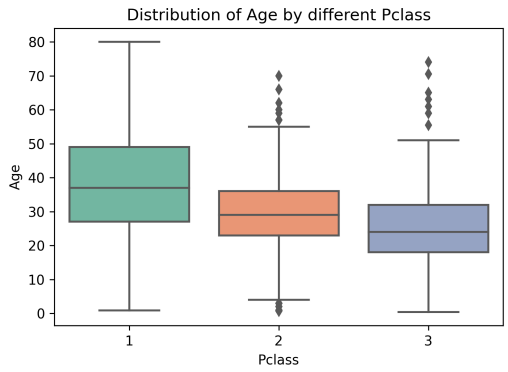
(b) Distribution of `Survived` over `Pclass`

Figure 5: Distribution of `Pclass` and its correlation with `Survived`

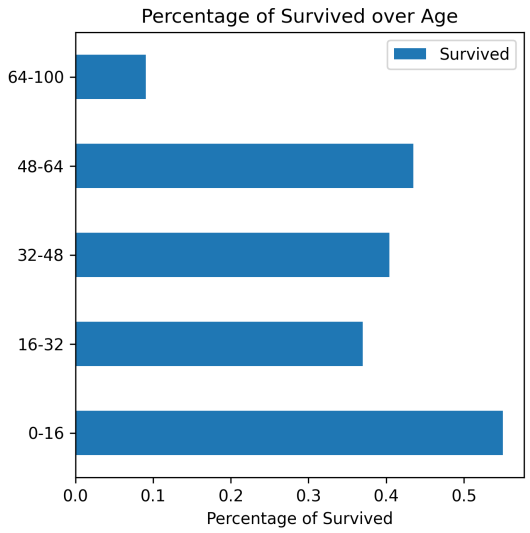
3. Therefore, this feature is kept and does not need to be one-hot encoded as the numerical values already represent the order of the classes.



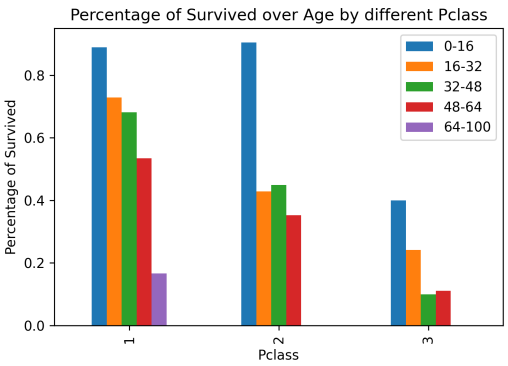
(a) Distribution of `Age`



(b) Distribution of `Age` over `Pclass`



(c) Distribution of `Age` over `Pclass`



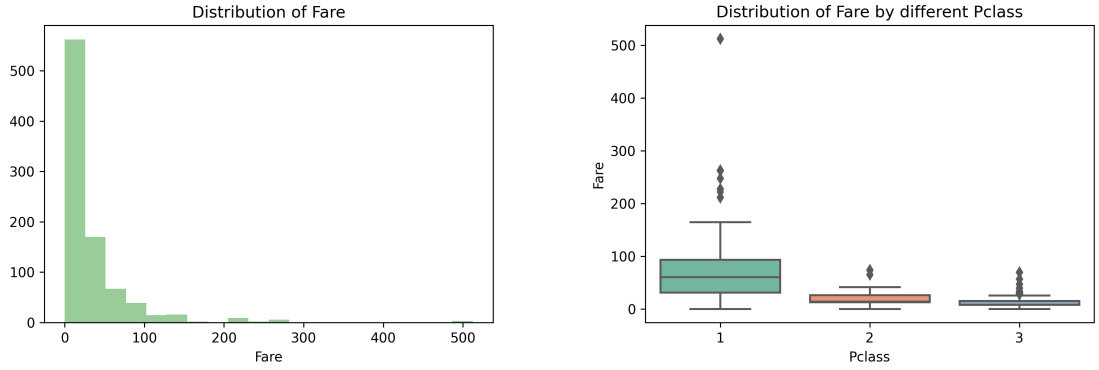
(d) Distribution of `Survived` of each age group over `Pclass`

Figure 6: Distribution of `Age` and its correlation with `Survived` and `Pclass`

Age: As illustrated in Figure 6, it was noted that there are 177 (19.865%) missing values within this

feature, with the majority of passengers falling within the age range of 16 to 35 years. Furthermore, the survival rate is notably higher among passengers under 16 years old compared to those over 16 years old, and that survival rates vary across different passenger classes within distinct age groups. As a result of these observations, several decisions were made regarding the feature. It was determined that the feature should be retained, and the missing values should be imputed using the median age within each passenger class. Additionally, to enhance the feature set, new features were introduced, including `AgeGroup` to categorize passengers by age groups, `IsChild` to identify whether a passenger is a child (under 16 years old), and `AgeXPclass` to represent the interaction between `AgeGroup` and `Pclass`. Lastly, the `Age` feature was scaled to a range of [0, 1] to ensure uniformity in its values.

SibSp & Parch: Both of these attributes provide information about the number of family members accompanying passengers on board the Titanic. The majority of passengers do not have any family members accompanying them during their voyage. In light of this observation, a decision has been made to introduce a new feature called `IsAlone` which will help determine whether a passenger is traveling alone or not. To streamline the dataset and eliminate redundancy, SibSp and Parch features has been removed, as their information can now be effectively captured by the newly created `IsAlone` feature.



(a) Distribution of `Fare`

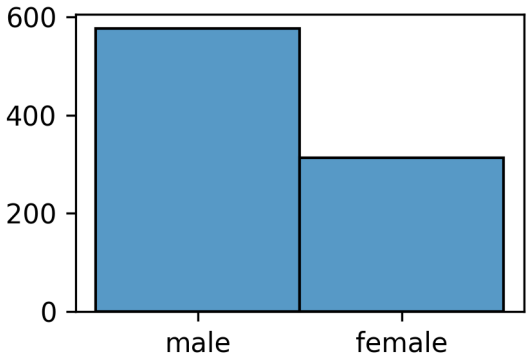
(b) Distribution of `Fare` over `Pclass`

Figure 7: Distribution of `Fare` and its correlation with `Pclass`

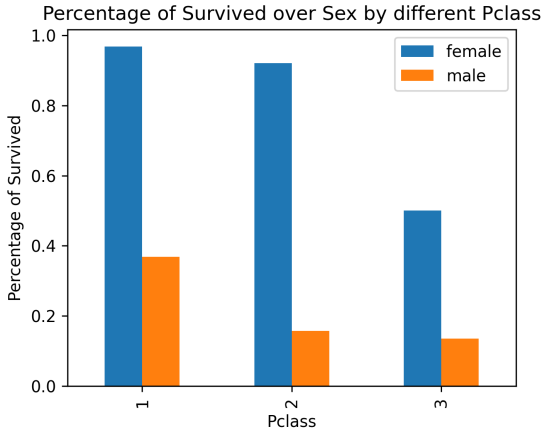
Fare: There is a single missing value in this feature, which will be filled with the median fare of the corresponding Pclass, as all Pclass categories exhibit skewed distribution in terms of fare. As Figure 7 shows, within the 1st class, there is an outlier with a fare value of 512.3292. To address this issue, a new feature will be introduced, `FareGroup`, to categorize passengers based on their fare. Lastly, the original `Fare` feature was dropped as part of the data preprocessing steps.

Sex: The examination of the `Sex` feature revealed several observations, as illustrated in Figure 8. Firstly, the passenger count for males surpasses that of females aboard the ship. Secondly, the distribution of sexes among passengers in various passenger classes is relatively consistent. Additionally, it was observed that female passengers exhibit a greater likelihood of survival. Consequently, I have chosen to make the following decisions regarding this feature: convert it into a binary feature named `IsFemale`, where '1' will signify female and '0' will represent male passengers. Simultaneously, the original `Sex` feature will be dropped.

Embarked: Regarding the `Embarked` feature, several observations were made. Firstly, there are two missing values within this attribute. Secondly, the majority of passengers boarded from Southampton. Additionally, it was noted that the survival rate of passengers who embarked from Cherbourg is notably higher compared to those who embarked from Queenstown and Southampton. However, this discrepancy is likely influenced by the fact that a significant portion of Cherbourg's passengers belong



(a) Distribution of Sex



(b) Distribution of Survived of each Pclass group over Sex

Figure 8: Distribution of Sex

to the 1st class. To address these observations, the missing values were filled using the most frequent value (mode) and proceed to one-hot encode this feature for further analysis.

1.2.2 Training and testing

For training and testing, I developed the following code to implement the logistic regression model without Scikit-learn:

```
class LogisticRegression:
    def __init__(self, lr=0.01, num_iter=10000):
        self.lr = lr
        self.num_iter = num_iter
        self.theta = None

    def __sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def __gradient(self, X, pred, y):
        return np.dot(X.T, (pred - y)) / y.size

    def fit(self, X, y):
        # Initialize the weights
        self.theta = np.zeros(X.shape[1])

        # Train the model using gradient descent
        for i in range(self.num_iter):
            z = np.dot(X, self.theta)
            pred = self.__sigmoid(z)
            gradient = np.dot(X.T, (pred - y)) / y.size
            self.theta -= self.lr * gradient

    def predict(self, X):
```

```
        z = np.dot(X, self.theta)
        pred = self.__sigmoid(z)
        return pred >= 0.5

model = LogisticRegression(lr=0.1, num_iter=150000)
model.fit(X_train, y_train)

y_pred = model.predict(X_val)
print(f"Accuracy: {accuracy_score(y_val, y_pred)}")
```

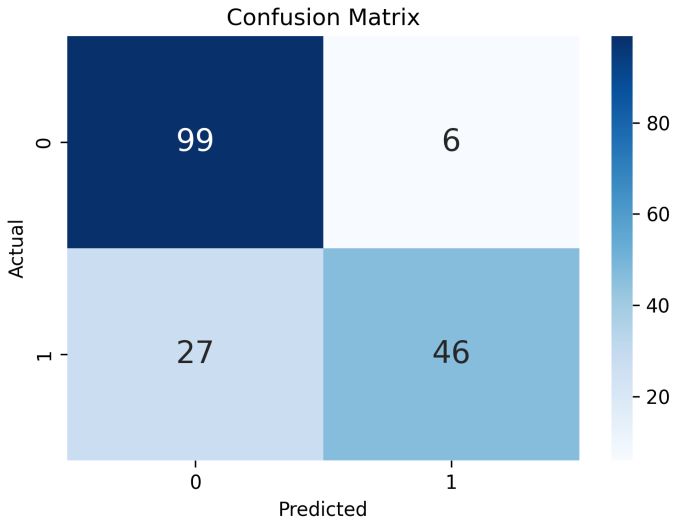


Figure 9: Confusion matrix of the logistic regression model

The accuracy on the validation set is 0.8146. Figure 9 illustrated the confusion matrix of the logistic regression model on the validation set. The precision is 0.8846, the recall is 0.6301 and the F1 score is 0.7360. The coefficients of the model are shown in Table 1.

Feature	**Coefficient**
Pclass	-9.937140
Age	31.988659
AgeXPclass	-0.863969
IsFemale	63.396897
IsChild	9.037811
IsAlone	3.696076
FareGroup	-0.611717
Embarked_C	20.067069
Embarked_Q	19.275919
Embarked_S	11.661376

Table 1: Coefficients of the logistic regression model

After training the model, the test set is loaded and preprocessed in the same way as the training set. Then, the model is used to predict the survival of passengers in the test set. The predictions are saved in a CSV file and submitted to Kaggle. The accuracy on the test set is 0.78708, which is ranked 1,331 out of 14,559+ teams on the leaderboard as in Figure 10.

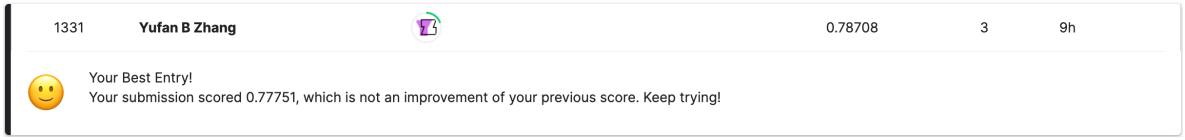


Figure 10: My Submission on Kaggle leaderboard

2 Written Exercises

2.1 Conceptual questions

a. Advantages and disadvantages of gradient descent

Advantage: Gradient descent is more general and can be used to optimize any objective function, whereas OLS is specific to the least squares objective function. In other words, if one wants to optimize a different objective function with no closed-form solution, gradient descent would be applicable.

Disadvantage: Gradient descent can be slower to converge than analytical formula approaches. Since the analytical formula such as OLS has a closed-form solution, it can be computed directly. However, gradient descent requires multiple iterations to converge to the optimal solution.

b. Regression by classification with discretization

It could be a possible approach to solve the regression problem by discretizing the continuous target variable into a finite number of bins and then solving a classification problem. However, this approach has several disadvantages, which often make it insufficient in practice.

- The accuracy of the classification problem is limited by the number of bins (i.e., the number of classes). If the interval within each bin is too large, the model will not be able to predict the target variable accurately.
- The discretization process may introduce computational complexity to the classification problem, especially when the number of bins is large.

c. One-vs-all classification vs. multi-class classification

Advantage: One-vs-all can be more useful when I have a reliable binary classifier. By applying one-vs-all, I can adapt the existing well-tuned binary classifier binary classifier to a multi-class problem.

Disadvantage: It can be computationally intensive if the number of classes is large. That is, I need to train k binary classifiers for k classes.

d. Polynomial regression

i. Dimension of the polynomial features

The dimension of the polynomial features that are needed to implement this model class is $O(p^d)$.

This is because for polynomial regression of degree p in d variables, the number of terms is given by the following formula:

$$\binom{d+p}{p}$$

That is, for each variable, I need to include all polynomial terms up to degree p.

ii. Computational complexity of polynomial least squares

The computational complexity of polynomial least squares is $O(p^{3d})$.

To solve the normal equation, I can find the least squares solution by computing the inverse of $X^T X$ and multiplying it with X^T. The complexity of inverting a matrix is $O(k^3)$ for a matrix of size $k \times k$. Since k is $O(p^d)$, the complexity becomes:

$$O((p^d)^3) = O(p^{3d})$$

Also, multiplying $X^T X$ itself is $O(n \cdot (p^d)^2)$. Therefore, the overall complexity is $O(p^{3d})$.

iii. Real-world settings

As the complexity of polynomial regression is $O(p^{3d})$, it might be **computationally expensive** to use it in real-world settings as the degree p or the number of attributes d increases. Also, mapping the data to a higher-dimensional space can lead to **overfitting**, which would result in poor generalization performance on unseen data. Moreover, the model might be too complex to **interpret**, which makes it less useful in practice.

Therefore, for datasets with many attributes or when a high polynomial degree is desired, it might be more practical to consider other regression methods or dimensionality reduction techniques, or to use regularization to control overfitting.

2.2 Analytical solution for Ordinary Least Squares

Given:

$$f_\theta(x^{(i)}) = \theta_0 + \theta_1 x^{(i)}$$
$$J(\theta_0, \theta_1) = \sum_{i=1}^{n} (y^{(i)} - f_\theta(x^{(i)}))^2$$

a. Partial derivatives

For θ_0:

$$\begin{aligned}\frac{\partial J}{\partial \theta_0} &= \frac{\partial}{\partial \theta_0} \sum_{i=1}^{n} (y^{(i)} - \theta_0 - \theta_1 x^{(i)})^2 \\ &= -2 \sum_{i=1}^{n} (y^{(i)} - \theta_0 - \theta_1 x^{(i)})\end{aligned}$$

For θ_1:

$$\begin{aligned}\frac{\partial J}{\partial \theta_1} &= \frac{\partial}{\partial \theta_1}\sum_{i=1}^{n}(y^{(i)} - \theta_0 - \theta_1 x^{(i)})^2 \\ &= -2\sum_{i=1}^{n}(y^{(i)} - \theta_0 - \theta_1 x^{(i)})x^{(i)}\end{aligned}$$

b. Using the normal equations

For θ_0, setting the gradient to zero:

$$\begin{aligned}&\sum_{i=1}^{n}(y^{(i)} - \theta_0 - \theta_1 x^{(i)}) = 0 \\ &\Rightarrow n\bar{y} = n\theta_0 + \theta_1\sum_{i=1}^{n}x^{(i)}\end{aligned}$$

For θ_1, setting the gradient to zero:

$$\begin{aligned}&\sum_{i=1}^{n}(y^{(i)} - \theta_0 - \theta_1 x^{(i)})x^{(i)} = 0 \\ &\Rightarrow \theta_1 = \frac{\sum_{i=1}^{n}(x^{(i)} - \bar{x})(y^{(i)} - \bar{y})}{\sum_{i=1}^{n}(x^{(i)} - \bar{x})^2}\end{aligned}$$

Using the derived value of θ_1, I can substitute it into the equation for θ_0:

$$\begin{aligned}n\bar{y} &= n\theta_0 + \frac{\sum_{i=1}^{n}(x^{(i)} - \bar{x})(y^{(i)} - \bar{y})}{\sum_{i=1}^{n}(x^{(i)} - \bar{x})^2}\sum_{i=1}^{n}x^{(i)} \\ \Rightarrow \theta_0 &= \bar{y} - \frac{\sum_{i=1}^{n}(x^{(i)} - \bar{x})(y^{(i)} - \bar{y})}{\sum_{i=1}^{n}(x^{(i)} - \bar{x})^2}\bar{x}\end{aligned}$$

Therefore, I can deduce:

$$\theta_0 = \bar{y} - \theta_1\bar{x}$$

c. Sum of residuals for optimal parameters

Given:

$$e^{(i)} = y^{(i)} - (\theta_0^* + \theta_1^* x^{(i)})$$

Summing over all residuals:

$$\sum_{i=1}^{n}e^{(i)} = \sum_{i=1}^{n}(y^{(i)} - \theta_0^* - \theta_1^* x^{(i)})$$

From the normal equations, the sum of residuals will be zero, which indicates that the best-fit line passes through the mean of the data points (i.e. $(\bar{x}, \bar{y})$). Moreover, the deviations above the regression line are balanced out by the deviations below the line on average.

2.3 Maximum Likelihood Estimation

a. Probabilistic model and learning paradigm

i. Probabilistic model:

Let p_i denote the denote the probability that the dice falls on side i. I can define out probabilistic model as the probabilities p_1, p_2, p_3, and p_4. Also, I have the following constraints:

$$\begin{aligned} p_1 + p_2 + p_3 + p_4 &= 1 \\ p_i &\geq 0 \quad \forall i \in \{1, 2, 3, 4\} \end{aligned}$$

ii. Learning paradigm:

The learning paradigm for this problem is **unsupervised learning** since I do not have any labels for the data (i.e., the true probabilities of each side of the dice). Therefore, I need to learn the parameters of the model from the data without any supervision.

b. Log-likelihood of the dataset

Each dice throw is independent and identically distributed (i.i.d.). Then, the likelihood of observing a sequence of n dice throws is given by the product of the probabilities of each individual dice throw:

$$\mathcal{L}(\theta) = \prod_{i=1}^{n} p_{x_i}$$

Taking the log on both sides, the log-likelihood is:

$$\log \mathcal{L}(\theta) = \sum_{i=1}^{n} \log p_{x_i}$$

Intuitive argument: An intuitive reason to optimize the log-likelihood instead of the likelihood is that taking the log transforms the product into a sum, which is numerically more stable and also makes the differentiation easier.

c. Maximum likelihood estimation

In the case of this dice toss problem, MLEs are the values of p_1, p_2, p_3, and p_4 that maximize the log-likelihood, which yields the frequencies of each side of the dice. That is:

$$\hat{p_i} = \frac{\text{number of times side } i \text{ appears}}{n}$$

Given the data in the table, we can compute the MLEs as follows:

$$\hat{p}_1 = 0$$
$$\hat{p}_2 = \frac{2}{8} = 0.25$$
$$\hat{p}_3 = \frac{2}{8} = 0.25$$
$$\hat{p}_4 = \frac{4}{8} = 0.5$$

This means, based on our observations, we estimate that the probabilities of the dice landing on sides 1, 2, 3, and 4 are 0, 0.25, 0.25, and 0.5 respectively.

d. Potential inaccuracies

Scenarios where this approach might yield inaccurate estimates:

- The number of dice throws is too small. In this case, the estimates might be inaccurate because the sample may not be representative of the true probabilities.
- The dice is not thrown randomly. In this case, the estimates might be inaccurate because the throws are not i.i.d.

Examples: If I throw the fair dice only 2 times and get the outcome sequence 1, 1, the MLE would be $\hat{p}_1 = 1$ and 0 for all other sides. However, this is clearly not a good estimate of the true probabilities. That is, the sample is too small to be representative of the true probabilities.