Q1: First, I drop the irrelevant data of the first row (question metadata) and the column "Duration (in seconds)" - as it was not directly relevant to the prediction survey respondent's current yearly compensation bucket). Most other features were retained because they could provide valuable insights into respondents' backgrounds. (Appendix - Figure 1a)

Then, I handle the missing value depending on the type of response: if it is a single-response question, missing values were imputed with mode to ensure missing values are filled to preserve the typical response distribution. The encoding of categorical columns were using label encoding, which assigns an integer value to each category and make it suitable for computational analysis. (Appendix - Figure 1b) If it is a multi-response question, I fill the unselected option with 0, indicating "not selected," while responses indicating selection were encoded as 1. This binary encoding approach reduces dimensionality and maintains the selection pattern for analysis. (Appendix - Figure 1c)

Q2: Using Lasso Regression, I identified and visualized the features with the highest importance. (Appendix - Figure 2) - Q4 and Q2 emerged as the most influential features in predicting yearly compensation, with the highest Lasso coefficient magnitudes. Q43 and Q40_selection_pattern and Q37_selection_pattern also showed strong associations (more than 0.4 in coefficient magnitude), suggesting that they might capture significant patterns related to compensation. Various other features displayed moderate importance with coefficient magnitude around 0.3. Q4 (Country of Residence) is a strong predictor capturing geographic salary variations driven by economic conditions across regions. Feature Q2 (Age) also holds high importance as it likely reflects career progression and accumulated experience.

Two new features were engineered for potential improvement in prediction - 'High Education Level': A binary feature indicating whether a respondent holds a graduate or professional degree. Another feature is 'Total Tools Used': A feature representing the total count of automated machine learning tools a respondent reported using, an aggregate indicator of tool usage intensity. These features were designed based on domain knowledge, hypothesizing that both education level and tool familiarity are positively associated with salary.

For feature selection, I applied Lasso Regression with Cross-Validation to identify the most relevant features and exclude non-informative ones. Lasso's regularization property penalizes less informative features, assigning them zero coefficients, thus selecting only the most impactful features. I used Lasso with a grid search over multiple alpha values to identify the optimal alpha parameter, which resulted in a refined feature set that balances complexity with predictive power. This selection process effectively reduced dimensionality while retaining the most predictive attributes, enhancing model interpretability and reducing overfitting risks.
Q3.1: Please refer to the notebook.
Q3.2: Based on the 10-fold cross-validation results with the default hyperparameter value C = 1.0, the model shows relatively consistent accuracy across the folds: (1) Average Accuracy Across 10 Folds: 0.3809 (2) Variance of Accuracy Across 10 Folds: 0.0002
The low variance indicates that the model's performance is stable across different subsets of the training data, suggesting it generalizes well to varying samples. (Appendix - Figure 3.2)

Q3.3: To analyze the bias-variance trade-off in the Ordinal Logistic Regression model, the hyperparameter *C* (inverse of regularization strength) is key. A higher *C* value reduces regularization, potentially lowering bias but increasing variance as the model fits more closely to the training data, risking overfitting. Conversely, a lower *C* value increases regularization, reducing variance but possibly introducing more bias by generalizing too much. Using grid search, I found the optimal *C* value to be 0.01, achieving an accuracy score of approximately 0.3858. (Appendix - Figure 3.3)
Q3.4: Scaling is necessary for our task because logistic regression, especially with regularization, is sensitive to the scale of features. Standardizing features ensures that each feature contributes equally to the model, preventing features with larger scales from dominating the model's weights. I applied StandardScaler to normalize features to a mean of 0 and a standard deviation of 1. Importantly, we scaled the training and test data separately: fitting the scaler only on the training set and then transforming both sets to prevent data leakage, ensuring consistent scaling without introducing bias from the test data.
Q4.1: Accuracy is not an ideal performance metric because it does not account for class imbalance or the importance of distinguishing between different classes. With ordinal or imbalanced classes (e.g. annual income often have more responses than very low extremes see Appendix - Figure 4.1)that most of participants fall in bucket 0), accuracy may misleadingly appear high by favoring the majority class, ignoring errors in the minority classes. Metrics like precision, recall, and F1-score are better suited as they offer insight into the model's performance across different classes, especially by evaluating true positives in specific classes. F1-score, in particular, balances precision and recall, helping to measure the model's ability to correctly classify each class without being biased by class frequency.
Q4.2: In an ordinal logistic regression model, several hyperparameters influence model performance, with C (inverse regularization strength) and max_iter (maximum iterations) being particularly impactful. The C parameter controls the strength of regularization, balancing the model's bias-variance trade-off. A higher C value reduces regularization, allowing the model to fit more closely to the training data, which can reduce bias but may increase variance, leading to potential overfitting. Conversely, lower C values increase regularization, which may help prevent overfitting by enforcing simpler model weights but could introduce more bias by overly smoothing the predictions. Regularization is essential in preventing the model from learning noise in the training data, especially with high-dimensional datasets, making C a crucial hyperparameter for fine-tuning.

The max_iter parameter defines the maximum number of iterations for the model to converge. If set too low, the model might stop before finding an optimal solution, especially when the data is complex or the optimization landscape is challenging. Thus, allowing a sufficient number of iterations helps ensure the solver converges to a stable solution.

For tuning, I conducted a grid search on C and max_iter using the F1-weighted metric from Question 4.1. This metric, which balances precision and recall, is well-suited for imbalanced datasets like ours. The optimal hyperparameters identified were C=10 and max_iter = 100, resulting in a weighted F1 score of approximately 0.251, indicating a balanced model performance across classes. (Appendix - Figure 4.2)

Q4.3: In comparing the feature importance graphs from my model ((Appendix - Figure 4.3) and Section 2, the results align closely, with Q4 (Country of Residence) and Q2 (Age) again emerging as the top predictors of yearly compensation. These findings underscore Q4's role in capturing salary differences based on geographic and economic factors, while Q2 likely reflects career stages and accumulated experience. Additional features such as Q43, Q40_selection_pattern, and Q37_selection_pattern also exhibit notable influence, suggesting these patterns may capture respondent preferences or practices that correlate with compensation. The consistency in feature ranking across analyses highlights the robustness of Q4 and Q2 as critical variables, while the moderate contributions of other patterns further enhance the model's ability to generalize compensation trends.
Q5.1: The model's performance on both the training and test sets is consistent (Appendix - Figure 5.1), with accuracy hovering around 0.39, indicating a fair level of generalization without significant overfitting or underfitting. However, precision and F1 scores are relatively low, particularly on the test set, where precision is around 0.21 and the F1 score is 0.26. This suggests that while the model correctly identifies positive cases, it struggles with balance, often misclassifying non-target classes. The low precision and F1 scores highlight areas for improvement, such as refining feature selection, increasing model complexity, or experimenting with advanced techniques to enhance class discrimination and prediction balance.
Q5.2: The model appears to be underfitting, as indicated by its relatively low accuracy, precision, and F1 scores across both the training and test sets. This consistency in performance between training and test sets suggests that the model does not have enough capacity or complexity to capture underlying patterns in the data, rather than overfitting to noise in the training set. To address this, enhancing feature engineering and exploring more complex models could significantly improve performance. For feature engineering, investigating the feature set to create new, more predictive variables could help capture intricate relationships. For example, interactions between features or aggregations might reveal underlying trends that a simpler model misses.

Additionally, implementing more complex models like ensemble methods, such as Random Forest or Gradient Boosting, could capture non-linear relationships in the data more effectively. These models often provide better performance by leveraging multiple decision trees, which helps the model adapt to varied patterns without strictly relying on linear relationships. By addressing the model's capacity limitations through refined features and enhanced model complexity, we can likely achieve better results on both training and test sets.
Q5.3: Please refer to Appendix - Figure 5.3
Q5.4: The dataset analysis and model performance reveal a significant class imbalance, with most responses clustered around lower target values. This imbalance likely drives the model to predict lower values more frequently, as reflected in the distributions for both training and test sets. The model's underfitting is evident, as it fails to capture higher target values or complex patterns, leading to low precision and F1 scores. The consistency of results across training and test sets indicates that the model lacks sufficient capacity to generalize effectively. Addressing the class imbalance through resampling or utilizing models designed for imbalanced data could enhance its predictive performance, particularly for underrepresented classes. Additionally, feature engineering might help in capturing more nuanced relationships within the data.

Appendix

Figure 1a: Clean data

```
def CleanData(df):
    # Drop the first row (question details)
    df.drop(df.index[0], inplace=True)

    #TODO: Drop other information too here if they are irrelevant to the task
    df.drop(columns=['Duration (in seconds)'], inplace=True)

    return df

salaries = CleanData(salaries)
salaries.shape
```

Figure 1b: Encode categorical features (single column responses)

```
###Encode categorical features (sinlge column responses)


    #TODO: Encode categorical features in the single column responses

    def ImputingAndEncodingSingleColFeatures(df):
        # Identify columns with single responses and explicitly include 'Q29_Encoded' (target)
        single_col_names = [col for col in df.columns if 'Q' in col and '_' not in col]

        single_col_names.append('Q29_Encoded')  # Ensure target is included

        # Impute missing values in single-column responses
        for col in single_col_names:
            if df[col].isnull().sum() > 0:
                # Fill categorical columns with the most frequent value (mode)
                df[col].fillna(df[col].mode()[0], inplace=True)

        # Encode categorical columns in single-column responses
        label_encoder = LabelEncoder()
        for col in single_col_names:
            if df[col].dtype == 'object':
                df[col] = label_encoder.fit_transform(df[col])

        return df

    # Apply the function to the salaries DataFrame
    salaries = ImputingAndEncodingSingleColFeatures(salaries)
```

Figure 1c: Handling categorical features (multi column responses)

```
###Handling categorical features (multi column responses)

import pandas as pd
import numpy as np

bucket = salaries['Q29_buckets']

def HandleMultiColResponsesAsPattern(df):
    # List of specific columns to exclude from being converted or dropped
    exclude_columns = ['Q29_Encoded']

    # Step 1: Identify and sort multi-column response groups by question number prefix (e.g., 'Q11_', 'Q44_')
    # Include only prefixes that have numbers after the initial letter(s)
    multi_col_prefixes = sorted(
        {
            col.split('_')[0]
            for col in df.columns if '_' in col and col.split('_')[0][1:].isdigit()
        },
        key=lambda x: int(x[1:])
    )  # Sort based on numeric part

    # Step 2: Process each group of multi-column responses
    for prefix in multi_col_prefixes:
        # Identify columns associated with the current question prefix
        multi_col_names = [col for col in df.columns if col.startswith(prefix + '_')]

        # Convert non-empty string responses to 1 (selected) and fill missing values with 0,
        # but skip columns in the exclude_columns list
        for col in multi_col_names:
            if col not in exclude_columns:  # Check if column is in exclude list
                df[col] = df[col].apply(lambda x: 1 if pd.notnull(x) and x != 'unknown' else 0)

        # Concatenate binary values into a single string pattern for each row
        df[prefix + '_selection_pattern'] = df[multi_col_names].astype(str).agg(''.join, axis=1)

        # Drop the original individual columns if only the concatenated pattern is needed,
        # but keep columns that are in the exclude_columns list
        columns_to_drop = [col for col in multi_col_names if col not in exclude_columns]
        df.drop(columns=columns_to_drop, inplace=True)


    return df

# Apply the function to the salaries DataFrame
salaries = HandleMultiColResponsesAsPattern(salaries)

# Print sample to verify the pattern columns
print("Sample of DataFrame after handling multi-column responses as selection patterns:")
print(salaries.head())
```

Figure 2: Top 10 Feature Importances from Lasso Regression

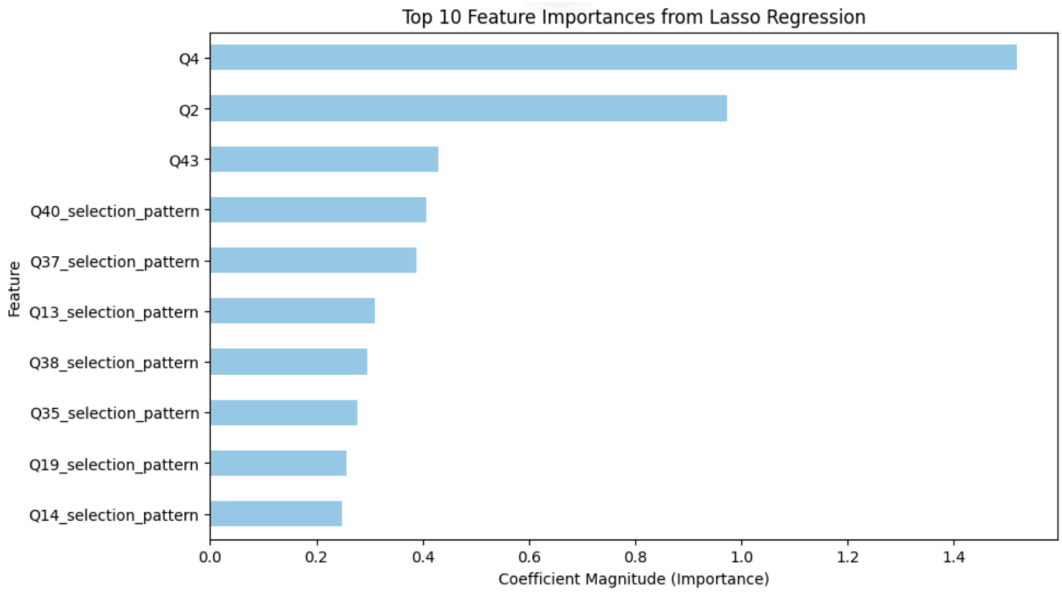

Figure 3.2: The average and variance of accuracy across folds

```
Evaluating OrdinalLogisticRegression with C=1.0
Accuracies for each fold: [0.38863287 0.38248848 0.38863287 0.35791091 0.3640553  0.39170507
 0.38248848 0.35637481 0.39076923 0.40615385]
Average accuracy across 10 folds: 0.38092118634054123
Variance of accuracy across 10 folds: 0.00023902720648628648

Summary of results:
C=1.0: Mean Accuracy=0.3809, Variance=0.0002
```

Figure 3.3: Search for best parameter values with ranging across values

```
Evaluating OrdinalLogisticRegression with C=0.01
Accuracies for each fold: [0.39324117 0.39170507 0.39170507 0.3655914  0.3655914  0.39784946
 0.39324117 0.3671275  0.39846154 0.41538462]
Average accuracy across 10 folds: 0.3879898381188704
Variance of accuracy across 10 folds: 0.00024787188868326894

Evaluating OrdinalLogisticRegression with C=0.1
Accuracies for each fold: [0.38863287 0.38248848 0.39016897 0.35791091 0.3625192  0.39324117
 0.38248848 0.359447   0.38769231 0.40769231]
Average accuracy across 10 folds: 0.38122816967978257
Variance of accuracy across 10 folds: 0.00023925315293606093

Evaluating OrdinalLogisticRegression with C=1
Accuracies for each fold: [0.38863287 0.38248848 0.38863287 0.35791091 0.3640553  0.39170507
 0.38248848 0.35637481 0.39076923 0.40615385]
Average accuracy across 10 folds: 0.38092118634054123
Variance of accuracy across 10 folds: 0.00023902720648628648

Evaluating OrdinalLogisticRegression with C=10
Accuracies for each fold: [0.38863287 0.38248848 0.38863287 0.35791091 0.3640553  0.39016897
 0.38248848 0.35637481 0.38923077 0.40615385]
Average accuracy across 10 folds: 0.38061373035566587
Variance of accuracy across 10 folds: 0.00023306213515500999
...
C=0.1: Mean Accuracy=0.3812, Variance=0.0002
C=1: Mean Accuracy=0.3809, Variance=0.0002
C=10: Mean Accuracy=0.3806, Variance=0.0002
C=100: Mean Accuracy=0.3806, Variance=0.0002
```

Figure 4.1: Training set distribution

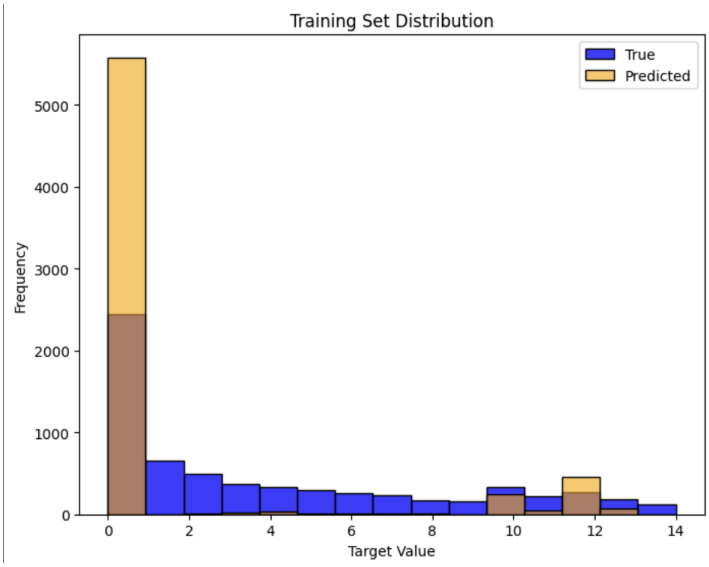

Figure 4.2: Grid search result

```python
def grid_search_ordinal_logistic(X_train, y_train):
    pipeline = Pipeline([
        ('scaler', StandardScaler()),
        ('ordinal_logreg', OrdinalLogisticRegression())
    ])

    param_grid = {
        'ordinal_logreg__C': [0.001, 0.01, 0.1, 1, 10],
        'ordinal_logreg__max_iter': [100, 500, 1000]
    }

    # Use 'f1_weighted' for F1 scoring that accounts for all class weights
    grid_search = GridSearchCV(pipeline, param_grid, scoring='f1_weighted', cv=5, n_jobs=-1)
    grid_search.fit(X_train, y_train)

    best_model = grid_search.best_estimator_.named_steps['ordinal_logreg']
    print("Best Parameters:", grid_search.best_params_)
    print("Best F1 Score:", grid_search.best_score_)

    return best_model, grid_search

# Example usage:
best_model, grid_search_results = grid_search_ordinal_logistic(X_train, y_train)
```

✓ 2.7s

```
Best Parameters: {'ordinal_logreg__C': 10, 'ordinal_logreg__max_iter': 100}
Best F1 Score: 0.25058236854808796
```

Figure 4.3: Top 10 Feature Importances in Ordinal Logistic Regression

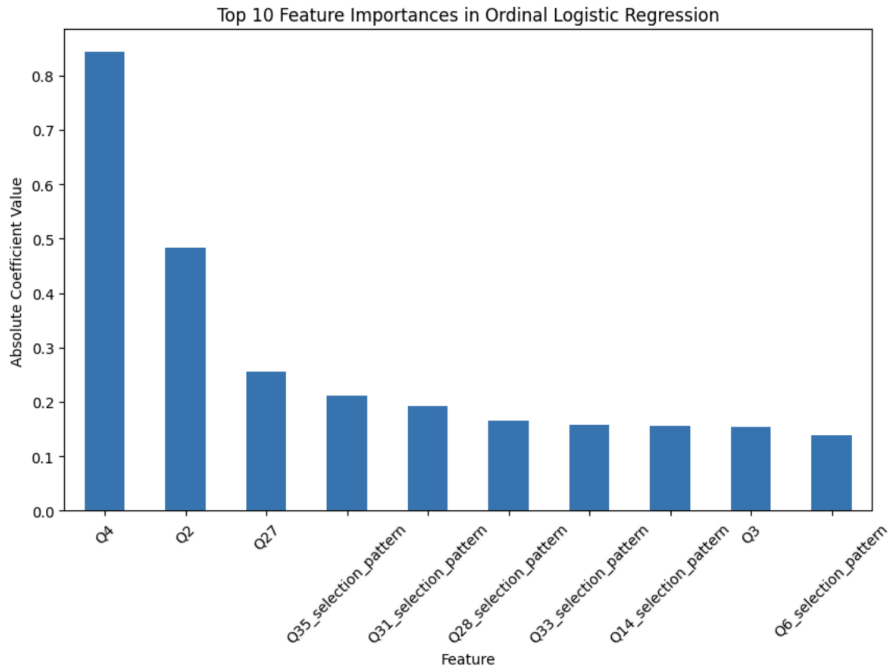

Figure 5.1: Performance on the test set vs. the training set

```
Performance on Training set:
  Accuracy: 0.3940
  Precision: 0.2777
  Recall: 0.3940
  F1 Score: 0.2621
--------------------------------------------------------
Performance on Test set:
  Accuracy: 0.3931
  Precision: 0.2131
  Recall: 0.3931
  F1 Score: 0.2609
--------------------------------------------------------
```

Figure 5.3: True vs Predicted Distributions of Target Values