STINTSY MP PRESENTATION

PREDICTING NATIONAL INCOME DECILE

TABLE OF CONTENTS

01 SUMMARY OF FINDINGS: TASK AND DATASET

02 EDA FINDINGS AND FEATURE ENGINEERING

03 MODEL TRAINING AND EVALUATION

04 CONCLUSION

SECTION 1

THE DATASET

The Family Income and Expenditure Survey (FIES) 2012, conducted by the Philippine Statistics Authority (PSA), provides comprehensive data on household income sources, demographics, and spending behaviors.

Two levels of income deciles were utilized in this survey - **national** and **regional**. The national income decile is obtained by ranking the weighted total family income of all sample families in the country from lowest to highest. Then these are compiled into ten groups. The first tenth, meaning those with the lowest income, is called the first decile; the second tenth, second decile and so on.

A similar procedure is implemented for the regional decile, except that what is arrayed is the weighted total family income of sample families in a given region.

SECTION 1

THE DATASET

This is a multi-class classification problem, where we aim to classify households into 10 different income groups (deciles) based on their financial and demographic characteristics.

Labels (Target Variable):

- NATDC (National Income Decile) - 1st tenth being poorest, last tenth being the richest

Input Features (Independent Variables):

- Demographic & Household Characteristics: Family size, sex, age, marital status, education level, employment details
- Income Sources: Wages, entrepreneurial income, cash assistance (domestic/abroad), pension, rental income, dividends
- Expenditure Patterns: Spending on food, housing, transportation, education, and healthcare

By training a classification model on this dataset, we aim to identify key determinants of income classification and evaluate the accuracy of the different machine learning models.

SECTION 2

UNDERSTANDING THE DATASET

1. The data set has 119 different features, many of which are elements of a certain set mainly TOINC, EAINC, T_TOTDIS, TOTEX, T_FOOD, T_FOOD_HOME, T_RENTVAL. This effectively means that if we needed, we could use these features to represent a large portion of the data. However this would lead to imbalances as they contribute a large amount when used in the models created.

2. The Question is structured where you are first given a Dichotomous Question (yes or no) followed by a Quantitative Question. This meant that if the dataset had blank spaces the respondent does not have anything for that feature meaning it has a value of 0 rather than null. But most importantly it meant that when cleaning the data we can not simply remove rows as part of our pre-processing activities nor give an assumed value to said rows.

SECTION 2

DATA CLEANING

The primary issue with the data was simply the missing spaces; these were neither null nor missing simply white spaces; aside from this the dataset did not require non specific cleaning for particular use in our models. While cleaning, we also used this time to confirm the data type of the features in the dataset which were all int64, while we could have covered more bases via forcing the categorical values into 0 or 1s it did not seem necessary enough for the entire data set to be changed. Once the cleaning was done we saved it into a data frame labeled CDATA.

Check for missing Values and duplicates

```
print("Number of data that is missing")
print(DATA.isnull().sum())

print ("\n")
print ("Number of Duplicates")
print(DATA.duplicated().sum())

DATA.replace(r'^\s', "0", regex=True, inplace=True)
DATA.fillna(0, inplace=True)
DATA.to_csv("/content/cDATA.csv", index=False)

print ("\n")
print("DATA START HERE")
file_path = "/content/cDATA.csv"
CDATA = pd.read_csv(file_path)

CDATA.head(100)
```

```
[6] print("Number of data that is missing")
    print(DATA.isnull().sum())

Number of data that is missing
W_REGN    0
W_OID     0
W_SHSN    0
W_HCN     0
URB       0
          ..
PCINC     0
NATPC     0
NATDC     0
REGDC     0
REGPC     0
Length: 119, dtype: int64
```

SECTION 2

EXPLORATORY DATA ANALYSIS

We used .describe from pandas to get the count, mean, standard deviation, minimum, quartiles (divided into four equal parts), and maximum values for each feature. After getting these to have a general feel of the whole dataset, this either helped prove some aspects in the data set that we already know and raised some issues.

```
[8] pd.set_option('display.max_rows', None)
    print(CDATA.describe().T)
```

	count	mean	std	min	\
W_REGN	40171.0	1.301989e+01	1.199555e+01	1.000000e+00	
W_OID	40171.0	4.210536e+09	2.285729e+09	1.010010e+08	
W_SHSN	40171.0	9.633666e+00	6.198442e+00	1.000000e+00	
W_HCN	40171.0	1.563602e+03	2.977364e+03	1.000000e+00	
URB	40171.0	1.617311e+00	4.860494e-01	1.000000e+00	
RSTR	40171.0	2.154728e+04	3.520981e+03	2.475000e+03	
PSU	40171.0	2.581237e+05	1.121433e+05	1.000100e+05	
BWEIGHT	40171.0	3.403304e+02	1.123779e+02	9.225000e+01	
RFACT	40171.0	5.333633e+02	2.099965e+02	1.261643e+02	
FSIZE	40171.0	4.699223e+00	2.194050e+00	1.000000e+00	
AGRI_SAL	40171.0	8.175135e+03	2.163931e+04	0.000000e+00	
NONAGRI_SAL	40171.0	8.946597e+04	1.602633e+05	0.000000e+00	
WAGES	40171.0	9.764110e+04	1.583316e+05	0.000000e+00	
NETSHARE	40171.0	1.572899e+03	1.062615e+04	0.000000e+00	
CASH_ABROAD	40171.0	2.313342e+04	8.052976e+04	0.000000e+00	
CASH_DOMESTIC	40171.0	1.011417e+04	2.459976e+04	0.000000e+00	
RENTALS_REC	40171.0	1.712772e+03	1.842484e+04	0.000000e+00	
INTEREST	40171.0	2.126626e+02	4.122624e+03	0.000000e+00	
PENSION	40171.0	7.471283e+03	4.833437e+04	0.000000e+00	

SECTION 2

EXPLORATORY DATA ANALYSIS

As stated earlier some features are extremely large and many of the categorical features have a big STD as well, the key ones being:

- Demographics
  - SEX - has an average is around 1.21, suggesting a coding system where male = 1 and female = 2
  - AGE - the average age of respondents is around 51 years old
  - MS - is ecoded such that (1) Single, (2) Married, (3) Widowed, (4) Separated, (5) Annuled, (6) Other
  - HGC - Highest grade completed

	count	mean	std	min
SEX	40171.0	1.213985e+00	4.101216e-01	1.000000e+00
AGE	40171.0	5.077100e+01	1.410533e+01	1.500000e+01
MS	40171.0	2.179109e+00	5.446988e-01	1.000000e+00
HGC	40171.0	3.748994e+02	1.904240e+02	0.000000e+00

SECTION 2

EXPLORATORY DATA ANALYSIS

As stated earlier some features are extremely large and many of the categorical features have a big STD as well, the key ones being:

- *Income and Wages*
  - WAGES - Average total is about 97,641
  - TOINC - sum of different sources of income
- *Housing and Assets*
  - TENURE - refers to house ownership
  - _QTY - number of items owned

	count	mean	std	min
WAGES	40171.0	9.764110e+04	1.583316e+05	0.000000e+00
TOINC	40171.0	2.176187e+05	2.536950e+05	5.201000e+03
TENURE	40171.0	1.895596e+00	1.547168e+00	1.000000e+00
RADIO_QTY	40171.0	4.617012e-01	5.358873e-01	0.000000e+00
TV_QTY	40171.0	8.464315e-01	6.541576e-01	0.000000e+00
CD_QTY	40171.0	5.398671e-01	5.951614e-01	0.000000e+00
STEREO_QTY	40171.0	1.847602e-01	4.126669e-01	0.000000e+00
REF_QTY	40171.0	3.849792e-01	5.352242e-01	0.000000e+00
WASH_QTY	40171.0	2.906823e-01	4.670018e-01	0.000000e+00
AIRCON_QTY	40171.0	1.077643e-01	4.102401e-01	0.000000e+00
CAR_QTY	40171.0	7.433223e-02	3.267984e-01	0.000000e+00
LANDLINE_QTY	40171.0	6.449927e-02	2.796688e-01	0.000000e+00
CELLPHONE_QTY	40171.0	1.706082e+00	1.543661e+00	0.000000e+00
PC_QTY	40171.0	2.033059e-01	5.378064e-01	0.000000e+00
OVEN_QTY	40171.0	1.418187e-01	3.616911e-01	0.000000e+00
MOTOR_BANCA_QTY	40171.0	1.202360e-02	1.302219e-01	0.000000e+00
MOTORCYCLE_QTY	40171.0	2.090812e-01	4.873538e-01	0.000000e+00

Another is that some features cannot be properly deciphered based on the data. A good example of this is HGC which has to say the least weird content all of which need a separate form to properly understand

SECTION 2

EXPLORATORY DATA ANALYSIS

Other interesting facts:

- There wasn't much standard deviation between regions however the highest came from region 13
- Most respondents came from Rural
- Majority are married
- Majority simply have a job

SECTION 2

EXPLORATORY DATA ANALYSIS

After this we further identified features primary results by dividing them into categorical/numerical as done to identify the data type of each feature particularly if they are performing categorical data or gathering various numerical data.

Numerical Variables

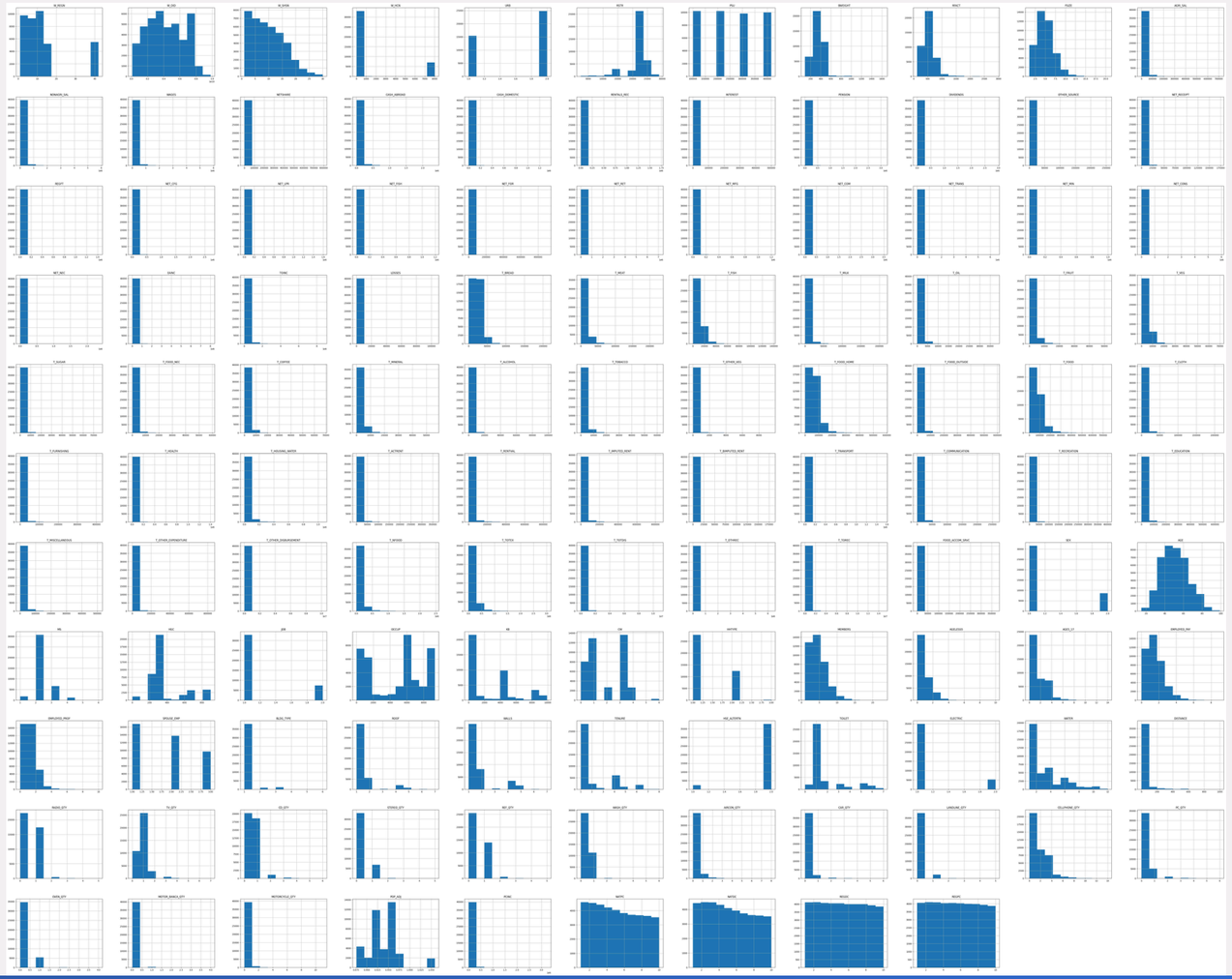
Variable	Variable	Variable	Variable
W_OID	W_HCN	BWEIGHT	RFACT
FSIZE	AGRI_SAL	NONAGRI_SAL	WAGES
NETSHARE	CASH_ABROAD	CASH_DOMESTIC	RENTALS_REC
INTEREST	PENSION	DIVIDENDS	OTHER_SOURCE
NET_RECIPT	REGFT	NET_CFG	NET_LPR
NET_FISH	NET_FOR	NET_RET	NET_MFD
NET_COM	NET_TRANS	NET_MIN	NET_CONS
NET_NEC	EAINC	TOINC	LOSSES
T_BREAD	T_MEAT	T_FISH	T_MILK
T_OIL	T_FRUIT	T_VEG	T_SUGAR
T_FOOD_NEC	T_COFFEE	T_MINERAL	T_ALCOHOL
T_TOBACCO	T_OTHER_VEG	T_FOOD_HOME	T_FOOD_OUTSIDE
T_FOOD	T_FURNISHING	T_HEALTH	T_HOUSING_WATER
T_ACTRENT	T_RENTVAAL	T_IMPUTED_RENT	T_TRANSPORT
T_COMMUNICATION	T_RECREATION	T_EDUCATION	T_MISCELLANEOUS
T_OTHER_EXPENDITURE	T_OTHER_DISBURSEMENT	T_NFOOD	T_TOTEX
T_TOTDIS	T_OTHEREC	FOOD_ACCOM_SRVC	AGE
MEMBERS	AGELESS5	AGE5_17	EMPLOYED_PAY
EMPLOYED_PROF	DISTANCE	RADIO_QTY	TV_QTY
CD_QTY	STEREO_QTY	WASH_QTY	AIRCON_QTY
CAR_QTY	LANDLINE_QTY	CELLPHONE_QTY	PC_QTY
OVEN_QTY	MOTOR_BANCA_QTY	POP_ADJ	PCINC

Categorical Variables

Variable	Variable	Variable	Variable
W_REGN	W_SHSN	URB	RSTR
PSU	SEX	MS	HGC
JOB	OCCUP	KB	CW
HHTYPE	SPOUSE_EMP	BLDG_TYPE	ROOF
WALLS	TENURE	HSE_ALTERTN	TOILET
ELECTRIC	WATER	NATPC	NATDC
REGDC	REGPC		

SECTION 2

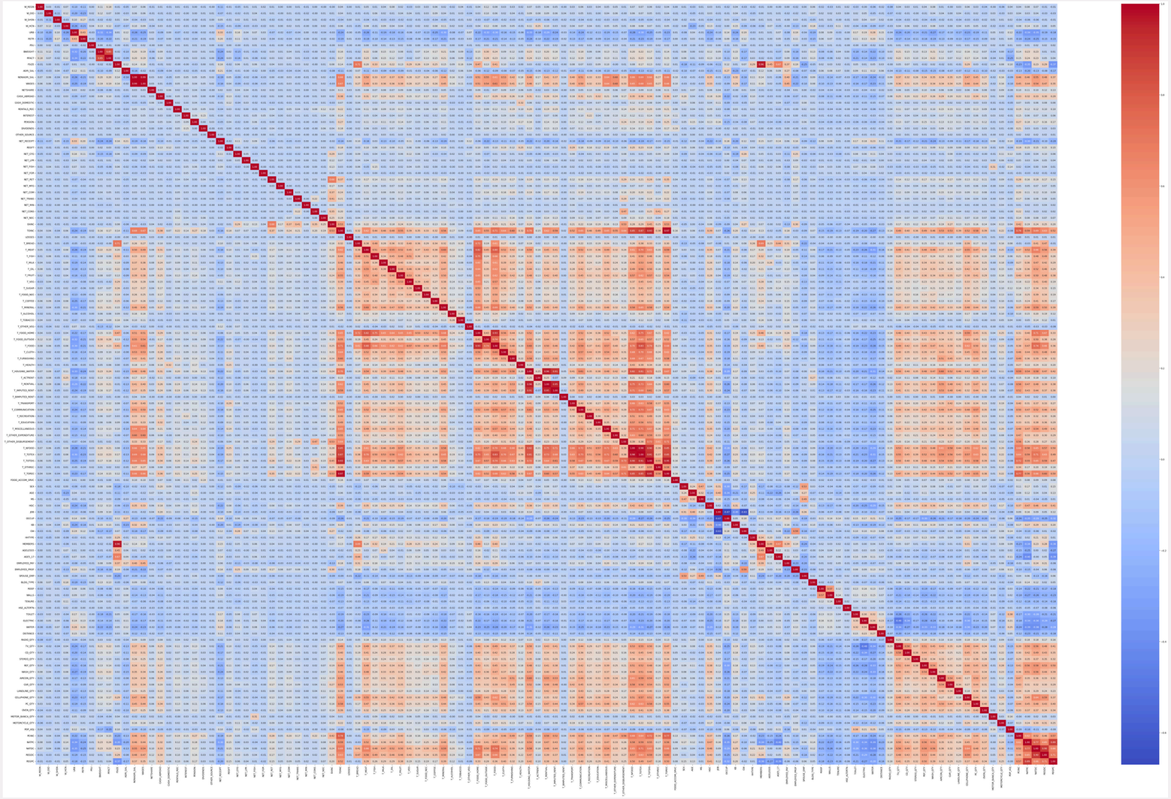
EXPLORATORY DATA ANALYSIS



SECTION 2

EXPLORATORY DATA ANALYSIS

After such, we proceeded to correlate each feature with each other with those at 0.8 and above:



1. BWEIGHT and RFACT → 0.85 - this is due to BWEIGHT being the assumed value of RFACT before final conclusion
2. FSIZE and MEMBERS → 0.97 - they both pertain to family size with FSIZE not including abroad members
3. NONAGRI_SAL and WAGES → 0.99 - majority dont have agricultural salary therefore they mostly only have wages
4. TOINC and T_NFOOD → 0.85 - part of T_TOTEX
5. TOINC and T_TOTEX → 0.87 - part of TOTDIS
6. TOINC and T_TOTDIS → 0.93 - Reverse of income
7. TOINC and T_TOREC → 0.97 - part of TOTDIS
8. T_MEAT and T_FOOD_HOME → 0.82 - Part of T_FOOD_HOME
9. T_FOOD_HOME and T_FOOD → 0.93 - T_FOOD_HOME is and element of T_FOOD
10. T_FOOD and T_TOTEX → 0.83 - part of TOTEX
11. T_HOUSING_WATER and T_RENTVAL → 0.96 - part of T_RENTVAL
12. T_HOUSING_WATER and T_IMPUTED_RENT → 0.91 - part of T_RENTVAL
13. T_HOUSING_WATER and T_NFOOD → 0.82- part of T_RENTVAL
14. T_HOUSING_WATER and T_TOTEX → 0.81 - since it is part of T_RENTVAL and T_RENTVAL is under T_TOTEX
15. T_RENTVAL and T_IMPUTED_RENT → 0.95 - part of T_RENTVAL
16. T_NFOOD and T_TOTEX → 0.98 - part of Other_Expenses which is under T_TOTEX
17. T_NFOOD and T_TOTDIS → 0.90 - part of Other_Expenses which is under T_TOTEX
18. T_NFOOD and T_TOREC → 0.85 - part of Other_Expenses which is under T_TOTEX
19. T_TOTEX and T_TOTDIS → 0.91- under TOTDIS
20. T_TOTEX and T_TOREC → 0.85 - under TOTDIS which is under T_TOREC (total receipts)
21. T_TOTDIS and T_TOREC → 0.95 - under T_TOREC
22. NATPC and REGPC → 0.89 - same thing just one is regional other is national
23. NATDC and REGDC → 0.90 - same thing just one is regional other is national

Note: if it says “part of” this typically means that it is the largest factor in the set we can view this in the extra tab under EDA

SECTION 2

EXPLORATORY DATA ANALYSIS

Finally, in feature selection we opted to select features greatly related to our target variable that being **NATDC** as such we typically selected values that had a **correlation value** of around **0.3 to 0.6.** This however, brought the issue that in feature importance, it tended to strongly value one feature over the other.

To address this, we used features with **lower correlation**. Which was done under the assumption that by adding data not fully relevant but somewhat logical we could force the models to look at the other features before making its prediction. For this we opted to add demographic features and asset ownership features to somewhat balance out the features

```
corr_matrix = CDATA.corr()
TFeature = "NATDC"

high_corr = corr_matrix[TFeature].abs() > threshold
high_corr = high_corr[high_corr.index != TFeature]

correlated_df = pd.DataFrame({
    "Feature": high_corr.index,
    "Correlation": corr_matrix.loc[high_corr.index, TFeature]
    }).reset_index(drop=True)

correlated_df_sorted = correlated_df.sort_values(by="Correlation", ascending=True)

print("All Correlation score compared to NATDC")
print(correlated_df_sorted[correlated_df_sorted["Correlation"].abs() >= -1].sort_values(by="Correlation", ascending=True))
```

```
84              MEMBERS     0.229500
8                 RFACT     0.232709
9                 FSIZE     0.234949
56             T_HEALTH     0.235007
87          EMPLOYED_PAY     0.235380
34                EAINC     0.239092
68  T_OTHER_DISBURSEMENT     0.278499
107         LANDLINE_QTY     0.286394
```

SECTION 2

EXPLORATORY DATA ANALYSIS

Finally, in feature selection we opted to select features greatly related to our target variable that being **NATDC** as such we typically selected values that had a **correlation value** of around **0.3 to 0.6.** This however, brought the issue that in feature importance, it tended to strongly value one feature over the other.

To address this, we used features with **lower correlation**. Which was done under the assumption that by adding data not fully relevant but somewhat logical we could force the models to look at the other features before making its prediction. For this we opted to add demographic features and asset ownership features to somewhat balance out the features

```
[14] FDATA = CDATA[[
        # Demographics
        "AGELESS5", "AGE", "SEX", "MEMBERS", "FSIZE", "HHTYPE", "BWEIGHT", "RFACT",

        # Economic & Income Indicators
        "NONAGRI_SAL", "PCINC", "WAGES", "EAINC", "EMPLOYED_PAY", "INTEREST",
        "DIVIDENDS", "PENSION", "OTHER_SOURCE", "RENTALS_REC", "NETSHARE",

        # Expenditures
        "T_MISCELLANEOUS", "T_EDUCATION", "T_HEALTH", "T_ALCOHOL", "T_TOBACCO",
        "T_FOOD_NEC", "T_FOOD_OUTSIDE", "T_TRANSPORT", "T_RECREATION", "T_COMMUNICATION",
        "T_HOUSING_WATER", "T_IMPUTED_RENT", "T_BIMPUTED_RENT", "T_ACTRENT", "T_RENTVAL",
        "T_OTHER_EXPENDITURE", "T_OTHER_DISBURSEMENT",

        # Asset Ownership
        "CELLPHONE_QTY", "TV_QTY", "REF_QTY", "WASH_QTY", "PC_QTY", "CAR_QTY",
        "MOTORCYCLE_QTY", "MOTOR_BANCA_QTY", "AIRCON_QTY", "STEREO_QTY", "RADIO_QTY",
        "LANDLINE_QTY", "OVEN_QTY",

        # Cash & Financial
        "CASH_DOMESTIC", "CASH_ABROAD",

        # Employment & Job Type
        "JOB", "FOOD_ACCOM_SRVC",

        # Other Key Factors
        "LOSSES", "NET_FISH", "NET_CONS", "NET_MIN", "NET_LPR", "NET_CFG",
        "NET_MFG", "NET_COM", "NET_RET", "NET_TRANS", "NET_NEC", "REGFT",
        "W_OID", "W_REGN", "W_HCN", "PSU", "REGPC", "HGC", "BLDG_TYPE",
        "T_SUGAR", "T_CLOTH", "T_VEG", "T_BREAD", "T_COFFEE", "T_FRUIT",
        "T_MILK", "T_FISH", "T_MINERAL", "CD_QTY"
    ]]
```

SECTION 2

EXPLORATORY DATA ANALYSIS

Feature Sets
1st: we selected features based on what we believed to predict the goal NATDC

2nd: we changed it since the results were weird with both train and test cases being too high, here we assumed there was some data leakage and utilized correlation as a means to select features.

3rd: Here we selected features with extremely high correlation, and were suggested to change them since they are too similar to the target features

4th: Due to the features having extremely biased feature importance we assumed that there was an issue with the feature possibly it being a source of data leakage (was a misunderstanding) and forcefully used features with little to no correlation to the target data in a attempt to equalize the feature importance (we didn't use any income feature and tried doing it with just _qty features and demographics

SECTION 2

EXPLORATORY DATA ANALYSIS

Feature Sets
5th: balanced out the features selected based on correlation from 0.3 to 0.6 as well as adding features with lower correlation to somewhat balance the feature importance

One thing that could help visualize the data more is if we had counted the number of entries greater than 0 to properly visualize the Dichotomous part of the questionnaire

SECTION 3

KNN MODEL

K-Nearest Neighbors (kNN) is a suitable model for predicting NATDC:

- It can handle non-linear relationships between the selected features and the target variable.
- Unlike parametric models, kNN does not make assumptions about the underlying data distribution, making it flexible for datasets with complex patterns.
- kNN is an intuitive and interpretable algorithm, as it classifies new data points based on the majority class of their nearest neighbors.
- Also, it it regarded as a good baseline for its straightforward implementation, making it a good starting point for comparing more complex models.
    - It requires minimal training, which helps quickly evaluate model feasibility and performance trends on a dataset.

SECTION 3

KNN MODEL

- We standardized the data after splitting the data into training testing sets to prevent data leakage and ensure it provides a reliable evaluation of the model's performance as there are features on different scales, which may affect the output.
- Across 10, iterations, it ranges **~0.32 to ~0.34** we get an overall average of **33.19%** accuracy score.
  - kNN model is not performing well on the given dataset
  - The model uses **k=5**

```python
    # standardize data
    sc = StandardScaler()
    X_train = sc.fit_transform(X_train)
    X_test = sc.transform(X_test)

    #initialize kNN Classifier
    knn_model = KNeighborsClassifier(n_neighbors=5)
    knn_model.fit(X_train, y_train)

    # make predictions on test data
    y_pred = knn_model.predict(X_test)

    accuracy = accuracy_score(y_test, y_pred)
    accuracy_scores.append(accuracy)

    print(f"Accuracy: {accuracy:.4f}")

avg_accuracy = np.mean(accuracy_scores)
print(f"\nFinal Average Accuracy over 10 Runs: {avg_accuracy:.4f}")
```

```
Training Iteration: 1
Accuracy: 0.3278
Training Iteration: 2
Accuracy: 0.3312
Training Iteration: 3
Accuracy: 0.3299
Training Iteration: 4
Accuracy: 0.3363
Training Iteration: 5
Accuracy: 0.3375
Training Iteration: 6
Accuracy: 0.3247
Training Iteration: 7
Accuracy: 0.3340
Training Iteration: 8
Accuracy: 0.3278
Training Iteration: 9
Accuracy: 0.3411
Training Iteration: 10
Accuracy: 0.3284

Final Average Accuracy over 10 Runs: 0.3319
```

SECTION 3

KNN MODEL

- With accuracy **0.33 (33%)**, this is quite low, suggesting that the model is struggling to make correct predictions.
- Class 0 and Class 9 Perform the Best
  - Class 0 has **precision = 0.50** and **recall = 0.80,** meaning it is correctly capturing many true positives.
  - Class 9 has **precision = 0.94** and **recall = 0.58**, indicating that when the model predicts class 9, it's usually correct, but it misses many actual class 9 instances.
- Worst Performing Classes (1-8)
  - Classes 1 to 8 have low precision, recall, and F1-scores.
- Issues suggest similarities in features between neighboring classes, leading to frequent misclassification.

```python
accuracy = accuracy_score(y_test, y_pred)
print(f"Model Accuracy: {accuracy:.2f}")

print(classification_report(y_test, y_pred))
cm = confusion_matrix(y_test, y_pred)
print('Confusion Matrix:\n', cm)
```

```
Model Accuracy: 0.33
              precision    recall  f1-score   support

           0       0.50      0.80      0.62       921
           1       0.25      0.36      0.30       899
           2       0.20      0.25      0.22       874
           3       0.19      0.20      0.19       835
           4       0.18      0.15      0.17       822
           5       0.21      0.16      0.18       787
           6       0.24      0.17      0.20       761
           7       0.32      0.24      0.27       726
           8       0.44      0.32      0.37       689
           9       0.94      0.58      0.72       721

    accuracy                           0.33      8035
   macro avg       0.35      0.32      0.32      8035
weighted avg       0.34      0.33      0.32      8035

Confusion Matrix:
 [[741 148  22   6   2   1   0   1   0   0]
 [402 328 132  28   8   1   0   0   0   0]
 [184 344 215  90  35   4   2   0   0   0]
 [ 92 230 246 165  72  23   7   0   0   0]
 [ 36 132 233 219 125  54  19   4   0   0]
 [ 17  67 151 184 171 129  53  15   0   0]
 [  6  33  56 110 174 193 127  54   7   1]
 [  1   8  26  55  82 144 191 171  46   2]
 [  0   4   4  13  20  61 117 228 218  24]
 [  1   1   0   1   1   2  13  54 228 420]]
```

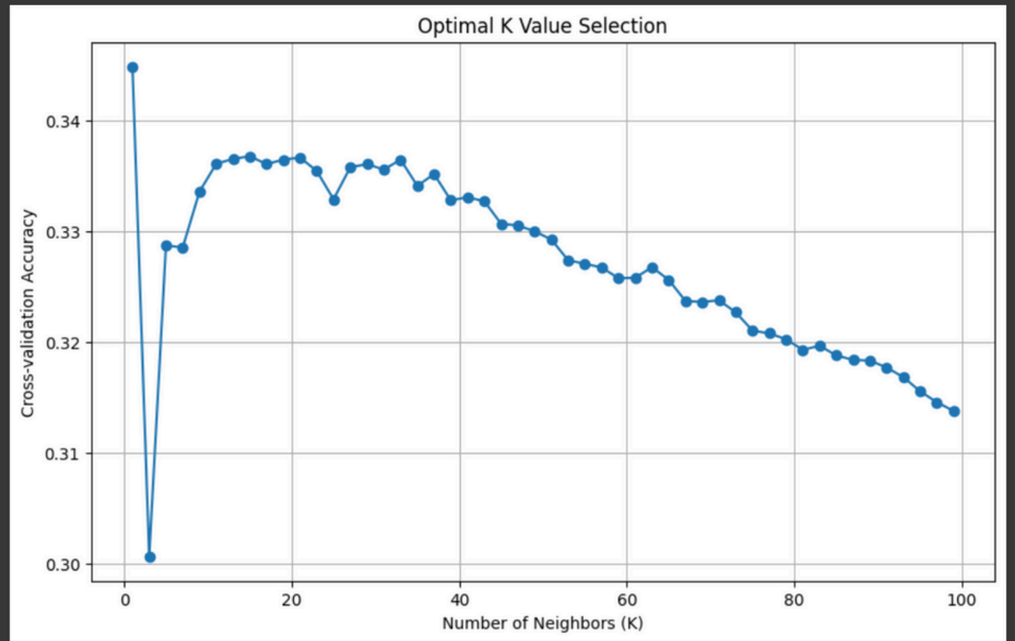
SECTION 3

KNN MODEL - TUNING

```
[22] # Cross Validation Score
     from sklearn.model_selection import cross_val_score
     k_folds = 5
     k_choices = [1, 3, 5, 8, 10, 12, 15, 20, 50, 100, 150, 200]

     scores = np.zeros((len(k_choices), 5))
```

- **Cross – Validation**
  - determine the optimal k value for the kNN model by dividing the dataset into multiple folds.
- **Grid Search**
  - test a range of k values, allowing us to identify the optimal parameter for the kNN model.
  - This suggest the optimal **k = 1**

```
# # Plot k values vs accuracy
plt.figure(figsize=(10, 6))
plt.plot(k_range, cv_scores, marker='o')
plt.xlabel('Number of Neighbors (K)')
plt.ylabel('Cross-validation Accuracy')
plt.title('Optimal K Value Selection')
plt.grid(True)
plt.show()
```



SECTION 3

KNN MODEL - TUNING

- **Best Parameters**
  - Best parameters: {'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'distance'} Best cross-validation accuracy: 0.3727
- Insights:
  - **Before tuning:** Average accuracy was **~0.3319.**
  - **After tuning:** Average accuracy is **~0.3607** (only a ~3% increase).
  - The best parameters found might not be significantly better than the default.
  - kNN might still struggle with the data structure, suggesting that tuning alone is not enough.
  - A different model might be necessary.

```python
from sklearn.model_selection import GridSearchCV
param_grid = {
    'n_neighbors': [optimal_k-2, optimal_k, optimal_k+2],
    'weights': ['uniform', 'distance'],
    'metric': ['euclidean', 'manhattan', 'minkowski ']
}

knn = KNeighborsClassifier()
grid_search = GridSearchCV(knn, param_grid, cv=5, scoring='accuracy', n_jobs=-1)
grid_search.fit(X_train, y_train)
```

```python
# # Print best parameters and score
print("\nBest parameters:", grid_search.best_params_)
print("Best cross-validation accuracy: {:.4f}".format(grid_search.best_score_))
```

```
Training Iteration: 1
Accuracy: 0.3591
Training Iteration: 2
Accuracy: 0.3704
Training Iteration: 3
Accuracy: 0.3637
Training Iteration: 4
Accuracy: 0.3654
Training Iteration: 5
Accuracy: 0.3685
Training Iteration: 6
Accuracy: 0.3635
Training Iteration: 7
Accuracy: 0.3609
Training Iteration: 8
Accuracy: 0.3571
Training Iteration: 9
Accuracy: 0.3679
Training Iteration: 10
Accuracy: 0.3306

Final Average Accuracy over 10 Runs: 0.3607
```

SECTION 3

KNN MODEL - INSIGHTS

- **Model Accuracy:**
  - ~37% testing accuracy after hyperparameter tuning, with optimal k values between 1–25.
  - Performance improved by 4.27% from the initial 33%, but still modest overall due to issues with class separability.
- **Sensitivity**
  - Highly sensitive to k values: Small values of k (1–25) provided better performance, capturing local patterns more effectively.
  - Larger k values (above 100) caused significant accuracy drops, highlighting the model's difficulty in handling complex, high-dimensional data.

SECTION 3

KNN MODEL - INSIGHTS

- **Behavioral Factors Matter Most:**
  - Income-spending behaviors and family size are strong predictors of income decile classification.
  - Larger families were more likely to fall into lower deciles, even with decent income, suggesting that lifestyle choices (e.g., spending habits) play a crucial role in income classification.
- **kNN Model Limitation:**
  - While kNN works well for small datasets and those with clear separations between classes, its scalability issues and sensitivity to hyperparameters make it less suitable for high-dimensional data like **NATDC.**
- **Next Steps**
  - Consider using more robust models that can better handle complex, high-dimensional data and provide more accurate class separability.
  - **Dimensionality reduction** techniques (e.g., PCA) might help reduce feature overlap and improve the model's performance.

SECTION 3

LOGISTIC REGRESSION

- Logistic Regression was selected to classify outcomes into categories, which aligns with our prediction goal.
- It’s easy to interpret and works well when features are scaled and classes are somewhat separable.
- Chosen as a baseline to test performance and see how well it handles challenges like outliers in the data.

SECTION 3

LOGISTIC REGRESSION

- solver='lbfgs'
  - Efficient for small/medium datasets; supports multiclass with L2 regularization.
- max_iter=1000
  - Maximum number of iterations to help the model converge.
- C=1.0 (default)
  - Controls regularization strength. Lower = more regularization.
- penalty='l2' (default)
  - L2 regularization helps prevent overfitting.
- multi_class='auto' (default)
  - Automatically picks the right strategy for binary or multiclass classification.

```python
for i in range(10):
    print(f"Training Iteration: {i+1}")

    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=i)

    # scaling the data
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    model = LogisticRegression(solver="lbfgs", max_iter=1000)
    model.fit(X_train_scaled, y_train)

    y_train_pred = model.predict(X_train_scaled)
    y_test_pred = model.predict(X_test_scaled)

    accuracy = accuracy_score(y_test, y_test_pred)
    test_accuracy_scores.append(accuracy)

    train_accuracy = accuracy_score(y_train, y_train_pred)
    train_accuracy_scores.append(train_accuracy)

    print(f"Train Accuracy: {train_accuracy:.4f} | Test Accuracy: {accuracy:.4f}")

avg_train_accuracy = np.mean(train_accuracy_scores)
avg_test_accuracy = np.mean(test_accuracy_scores)
```

SECTION 3

LOGISTIC REGRESSION

Across 10, iterations, we get an overall average of **81.57%** accuracy score.

Since the train and test accuracies are only **one percent apart,** this suggests that the model generalizes well to new data.

```
Training Iteration: 1
Train Accuracy: 0.8289 | Test Accuracy: 0.8158
Training Iteration: 2
Train Accuracy: 0.8272 | Test Accuracy: 0.8154
Training Iteration: 3
Train Accuracy: 0.8272 | Test Accuracy: 0.8132
Training Iteration: 4
Train Accuracy: 0.8271 | Test Accuracy: 0.8102
Training Iteration: 5
Train Accuracy: 0.8277 | Test Accuracy: 0.8147
Training Iteration: 6
Train Accuracy: 0.8269 | Test Accuracy: 0.8159
Training Iteration: 7
Train Accuracy: 0.8268 | Test Accuracy: 0.8182
Training Iteration: 8
Train Accuracy: 0.8249 | Test Accuracy: 0.8119
Training Iteration: 9
Train Accuracy: 0.8268 | Test Accuracy: 0.8187
Training Iteration: 10
Train Accuracy: 0.8281 | Test Accuracy: 0.8227

Final Average Train Accuracy over 10 Runs: 0.8272
Final Average Test Accuracy over 10 Runs: 0.8157
```

SECTION 3

LOGISTIC REGRESSION

Classes 1, 2, and 3 had **lower precision and recall (0.72–0.77)**, with frequent misclassifications, particularly between class 1 and class 2, and class 3 and class 2.

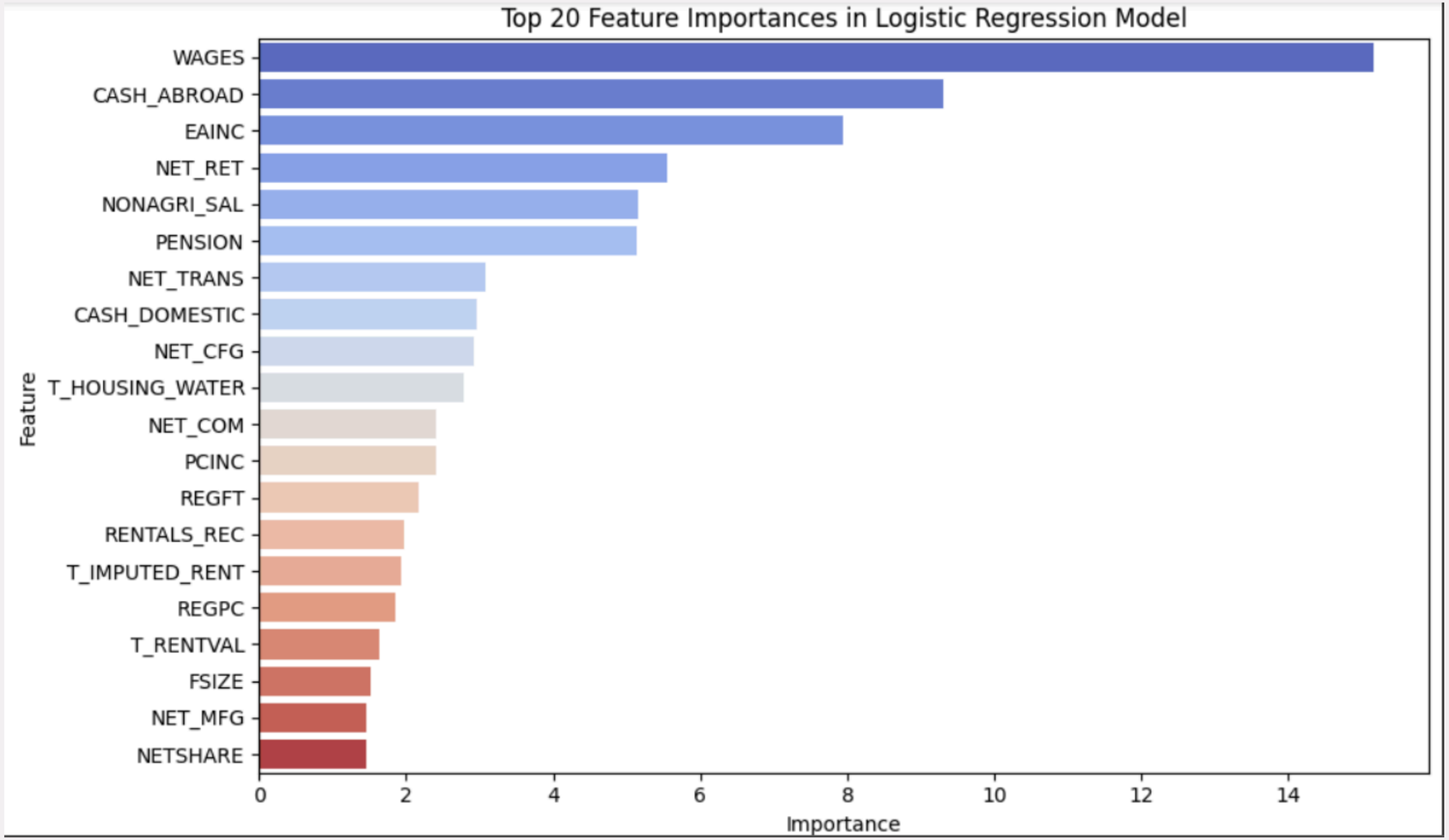
On the other hand, **Class 9** had the **highest performance (precision 0.99, recall 0.98)**, followed by **Class 8 (precision 0.95, recall 0.92)**, while **Classes 0 and 7** also showed strong results **(precision and recall > 0.88)**.

These issues suggest **similarities in features between neighboring classes**, with the model performing more reliably for **classes with clearer feature distinctions.**

```
Model Accuracy: 0.82
              precision    recall  f1-score   support

           0       0.90      0.88      0.89       921
           1       0.75      0.77      0.76       899
           2       0.72      0.73      0.72       874
           3       0.71      0.73      0.72       835
           4       0.75      0.73      0.74       822
           5       0.79      0.79      0.79       787
           6       0.86      0.84      0.85       761
           7       0.89      0.91      0.90       726
           8       0.95      0.92      0.93       689
           9       0.99      0.98      0.98       721

    accuracy                           0.82      8035
   macro avg       0.83      0.83      0.83      8035
weighted avg       0.82      0.82      0.82      8035

Confusion Matrix:
 [[807 114   0   0   0   0   0   0   0   0]
 [ 90 692 117   0   0   0   0   0   0   0]
 [  0 115 639 117   3   0   0   0   0   0]
 [  0   4 133 607  91   0   0   0   0   0]
 [  0   0   3 134 598  87   0   0   0   0]
 [  0   1   0   1 104 623  58   0   0   0]
 [  0   0   0   0   0  83 642  36   0   0]
 [  0   0   0   0   0   0  45 659  22   0]
 [  0   0   0   0   0   0   0  44 636   9]
 [  0   0   0   0   0   0   0   0  14 707]]
```

SECTION 3

LOGISTIC REGRESSION



SECTION 3

LOGISTIC REGRESSION

```python
param_grid = {
    'C': [0.001, 0.01, 0.1, 1, 10, 100],
    'penalty': ['l2'],
    'solver': ['lbfgs', 'newton-cg', 'sag'],
    'multi_class': ['ovr', 'multinomial'],
    'class_weight': ['balanced']
}

grid_search = GridSearchCV(
    LogisticRegression(max_iter=1000, random_state=42),
    param_grid,
    cv=5,
    scoring=['accuracy', 'f1_weighted', 'precision_weighted', 'recall_weighted'],
    refit='f1_weighted',
    n_jobs=-1,
    verbose=1
)
grid_search.fit(X_train_scaled, y_train)

print("Best Parameters:", grid_search.best_params_)
print("Best Cross-Validation Accuracy:", grid_search.best_score_)

best_model = grid_search.best_estimator_
y_test_pred = best_model.predict(X_test_scaled)

test_accuracy = accuracy_score(y_test, y_test_pred)
print("Test Set Accuracy:", test_accuracy)
```

The hyperparameter tuning process was too **time-consuming** and couldn't be completed in time for this analysis.

To optimize the tuning process:

- use randomized search instead of grid search

Since this code took too long to run, I tinkered with the code a bit

To optimize Logistic Regression:

- decrease regularization, but this may cause overfitting

SECTION 3

LOGISTIC REGRESSION

Adjusted the C value

- smaller c = stronger regularization
- larger c = weaker regularization

Larger C values allows the model to fit the training data, but this may potentially cause overfitting

```
Final Average Train Accuracy over 10 Runs: 0.9135
Final Average Test Accuracy over 10 Runs: 0.9032
```

C = 10, accuracy score increased by 8.75%

```
Final Average Train Accuracy over 10 Runs: 0.9209
Final Average Test Accuracy over 10 Runs: 0.9113
```

C = 15, accuracy score increased by 9.56%

```
Final Average Train Accuracy over 10 Runs: 0.9250
Final Average Test Accuracy over 10 Runs: 0.9151
```

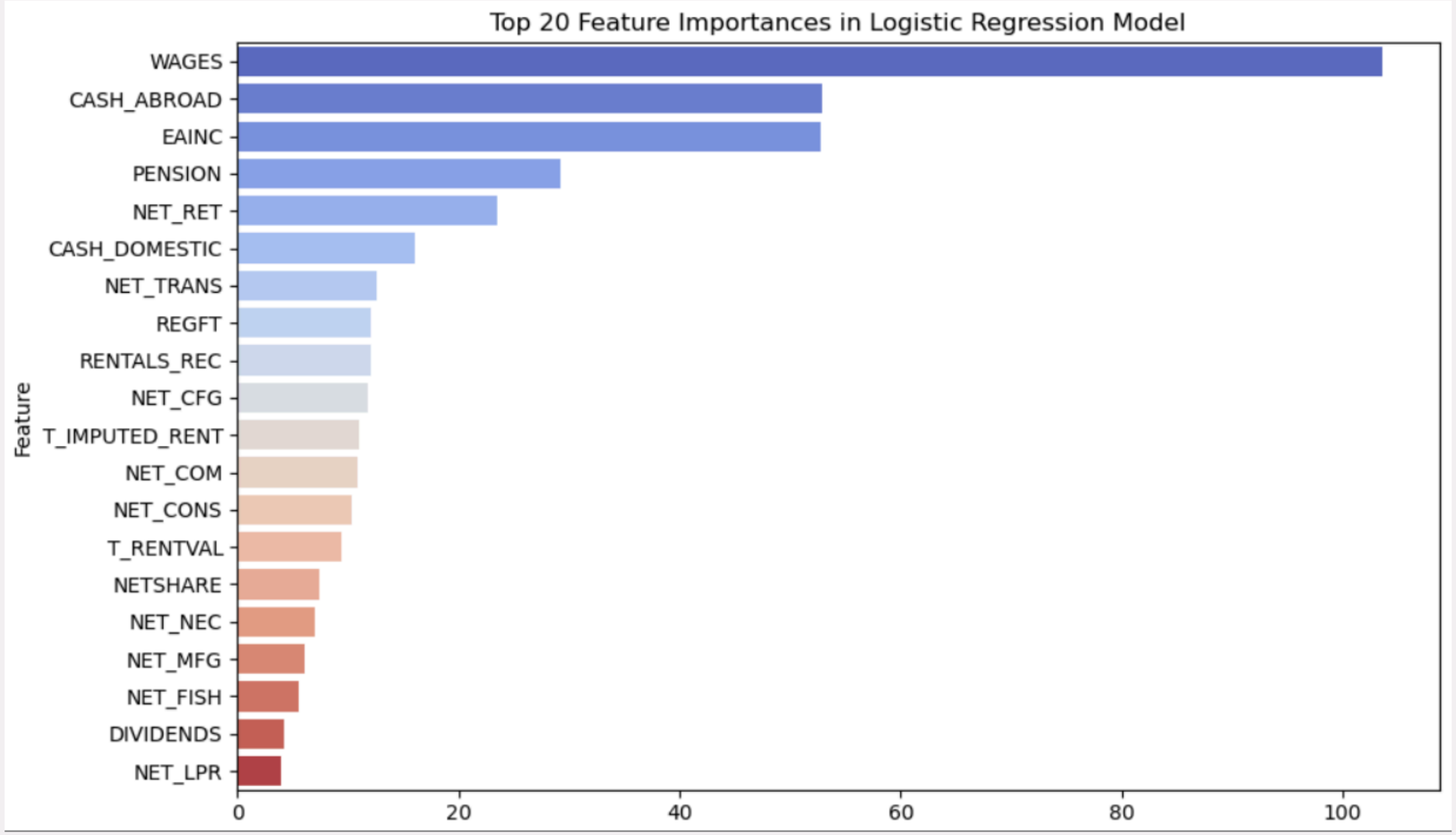
C = 20, accuracy score increased by 9.94%

```
Final Average Train Accuracy over 10 Runs: 0.9330
Final Average Test Accuracy over 10 Runs: 0.9233
```

C = 50, accuracy score increased by 10.76%

SECTION 3

LOGISTIC REGRESSION



SECTION 3

LOGISTIC REGRESSION

- Higher overall accuracy **(93% vs. 82%)**
- More balanced precision and recall across classes:
- Particularly strong improvement for lower-performing classes (e.g., class 3), leading to more reliable predictions across all income deciles

```
Model Accuracy: 0.93
              precision    recall  f1-score   support

           0       0.96      0.96      0.96       921
           1       0.90      0.90      0.90       899
           2       0.88      0.86      0.87       874
           3       0.86      0.88      0.87       835
           4       0.89      0.89      0.89       822
           5       0.92      0.92      0.92       787
           6       0.96      0.95      0.95       761
           7       0.97      0.97      0.97       726
           8       0.98      0.98      0.98       689
           9       0.99      0.99      0.99       721

    accuracy                           0.93      8035
   macro avg       0.93      0.93      0.93      8035
weighted avg       0.93      0.93      0.93      8035

Confusion Matrix:
 [[881  40   0   0   0   0   0   0   0   0]
 [ 40 810  49   0   0   0   0   0   0   0]
 [  0  52 753  69   0   0   0   0   0   0]
 [  0   0  48 737  50   0   0   0   0   0]
 [  0   2   0  52 735  33   0   0   0   0]
 [  1   0   1   0  43 727  15   0   0   0]
 [  0   0   0   0   0  30 720  11   0   0]
 [  0   0   0   0   0   0  13 704   9   0]
 [  0   0   0   0   0   0   0   9 673   7]
 [  0   0   0   0   0   0   0   0   5 716]]
```

SECTION 3

LOGISTIC REGRESSION - INSIGHTS

- Logistic Regression performed moderately well, with accuracy scores ranging from 82.57% to 92.33%.
- The model captured meaningful patterns in the data, but performed better with classes that have more distinct features (Classes 0, 8, 9).
- Classes with similar features (Classes 1, 2, 3) were harder to classify accurately.
- Weakening the model's regularization (by increasing the C value to 10, 15, 20, and 50) led to better results, with a significant increase in accuracy.

SECTION 3

XGBOOST

XGBoost is an optimized distributed gradient boosting library designed to be highly efficient, flexible and portable. It uses an ensemble of decision trees that are constructed sequentially to minimize the residual errors of the previous trees.

The XGBoost model was selected due its capabilities:

1. It can handle high-dimensional, tabular data efficiently using tree-based additive boosting.
2. The model is immune to feature scaling since trees are split based on thresholds, not distances
3. Handles sparse or missing values using Default Direction in creating the trees

The model was trained under 10 iterations.

SECTION 3

XGBOOST

The following hyperparameters were used:

- Number of estimators (n_estimators): 200 (controls the number of boosting rounds)
- Learning rate (learning_rate): 0.05 (moderate step size to balance convergence and performance)
- Maximum tree depth (max_depth): 6 (prevents excessive model complexity and overfitting)
- Column sampling (colsample_bytree): 0.8 (reduces overfitting by selecting a subset of features)
- Subsampling (subsample): 0.8 (ensures diversity in training samples)
- Random state (random_state): Set per iteration for reproducibility

```
[ ] accuracy_scores = []
    train_accuracy_scores = []
    iter = 10

    for i in range(iter):
        print(f"Training Iteration: {i+1}")

        X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=i)

        # model init
        xgb_model = xgb.XGBClassifier(
            n_estimators=200, learning_rate=0.05, max_depth=6,
            colsample_bytree=0.8, subsample=0.8, random_state=i
        )
        xgb_model.fit(X_train, y_train)

        y_train_pred = xgb_model.predict(X_train)
        y_test_pred = xgb_model.predict(X_test)

        train_accuracy = accuracy_score(y_train, y_train_pred)
        test_accuracy = accuracy_score(y_test, y_test_pred)

        accuracy_scores.append(test_accuracy)
        train_accuracy_scores.append(train_accuracy)

        # per iteration accuracy
        print(f"Training Accuracy: {train_accuracy:.4f}")
        print(f"Test Accuracy: {test_accuracy:.4f}")
        print("-" * 50)

    avg_test_accuracy = np.mean(accuracy_scores)
    avg_train_accuracy = np.mean(train_accuracy_scores)
    print(f"\nFinal Average Training Accuracy over 10 Runs: {avg_train_accuracy:.4f}")
    print(f"Final Average Test Accuracy over 10 Runs: {avg_test_accuracy:.4f}")
```

SECTION 3

XGBOOST

Across 10 iterations...

Overall Average: 90.85%

Has a small generalization gap of 5.96% from the training accuracy of 96.81%, indicating minimal overfitting.

```
--------------------------------------------------
Training Iteration: 7
Training Accuracy: 0.9613
Test Accuracy: 0.9047
--------------------------------------------------
Training Iteration: 8
Training Accuracy: 0.9597
Test Accuracy: 0.9108
--------------------------------------------------
Training Iteration: 9
Training Accuracy: 0.9590
Test Accuracy: 0.9101
--------------------------------------------------
Training Iteration: 10
Training Accuracy: 0.9616
Test Accuracy: 0.9040
--------------------------------------------------

Final Average Training Accuracy over 10 Runs: 0.9601
Final Average Test Accuracy over 10 Runs: 0.9085
```

SECTION 3

XGBOOST

Error analysis showed that **1,157 instances were misclassified**, with **Classes 3, 5, and 6 being the most problematic** due to lower recall and F1-scores (~0.87), and frequent misclassification with adjacent deciles.

In contrast, **Classes 0 and 9 performed the best**, likely due to more distinct feature profiles.

These challenges point to feature similarity among neighboring deciles, possible class imbalance, and limited model expressiveness for mid-range income groups.

```
Classification Report:
              precision    recall  f1-score   support

           0       0.94      0.98      0.96      1344
           1       0.88      0.91      0.89      1355
           2       0.88      0.87      0.88      1374
           3       0.88      0.87      0.87      1268
           4       0.86      0.88      0.87      1240
           5       0.88      0.87      0.87      1145
           6       0.90      0.88      0.89      1107
           7       0.91      0.90      0.90      1093
           8       0.94      0.92      0.93      1066
           9       0.99      0.96      0.98      1060

    accuracy                           0.90     12052
   macro avg       0.91      0.90      0.90     12052
weighted avg       0.90      0.90      0.90     12052


Per-class Accuracy:
Class 0: 0.9777
Class 1: 0.9129
Class 2: 0.8748
Class 3: 0.8667
Class 4: 0.8806
Class 5: 0.8734
Class 6: 0.8753
Class 7: 0.8966
Class 8: 0.9221
Class 9: 0.9613

Number of misclassified samples: 1157
```

SECTION 3

XGBOOST - TUNING

Optuna is an open source hyperparameter optimization framework to automate hyperparameter search.

This was used to tune critical XGBoost parameters such as max_depth, learning_rate, n_estimators, colsample_bytree, and subsample.

It searches the parameter space intelligently to maximize the macro F1-score, using 50 optimization trials to find the best-performing combination.

To ensure balanced class representation during evaluation, StratifiedKFold (10-fold) was used.

```python
def objective(trial):
    max_depth = trial.suggest_int("max_depth", 3, 10)
    learning_rate = trial.suggest_float("learning_rate", 0.01, 0.3, log=True)
    n_estimators = trial.suggest_int("n_estimators", 100, 500, step=50)
    colsample_bytree = trial.suggest_float("colsample_bytree", 0.5, 1.0)
    subsample = trial.suggest_float("subsample", 0.5, 1.0)

    model = xgb.XGBClassifier(
        max_depth=max_depth,
        learning_rate=learning_rate,
        n_estimators=n_estimators,
        colsample_bytree=colsample_bytree,
        subsample=subsample,
        random_state=42,
        eval_metric="mlogloss"
    )

    skf = StratifiedKFold(n_splits=10, shuffle=True, random_state=42)
    f1_macro = cross_val_score(model, X, y, cv=skf, scoring="f1_macro").mean()
    accuracy = cross_val_score(model, X, y, cv=skf, scoring="accuracy").mean()

    trial.set_user_attr("accuracy", accuracy)

    return f1_macro

study = optuna.create_study(direction="maximize")
study.optimize(objective, n_trials=50)

best_accuracy = study.best_trial.user_attrs["accuracy"]

print("Best hyperparameters:", study.best_params)
print(f"Best F1-score: {study.best_value:.4f}")
print(f"Best accuracy: {best_accuracy:.4f}")
```

SECTION 3

XGBOOST - POST TUNING

Best Hyperparameters:

- max_depth: 8
- learning_rate: 0.058
- n_estimators: 450
- colsample_bytree: 0.656
- subsample: 0.955

Test Accuracy: 92.2%

Training Accuracy: 100%

The 100% training accuracy is suspicious, but since the test accuracy is high as well, this might suggest mild overfitting.

```
xgb_model = xgb.XGBClassifier(
    n_estimators=450, learning_rate=0.05819068217437232, max_depth=8,
    colsample_bytree=0.6567234139229252, subsample=0.9558111605361372, random_state=i
)
xgb_model.fit(X_train, y_train)
```

```
Test Accuracy: 0.9187
--------------------------------------------------
Training Iteration: 8
Training Accuracy: 1.0000
Test Accuracy: 0.9233
--------------------------------------------------
Training Iteration: 9
Training Accuracy: 1.0000
Test Accuracy: 0.9244
--------------------------------------------------
Training Iteration: 10
Training Accuracy: 1.0000
Test Accuracy: 0.9170
--------------------------------------------------

Final Average Training Accuracy over 10 Runs: 1.0000
Final Average Test Accuracy over 10 Runs: 0.9215
```

SECTION 3

XGBOOST - POST TUNING

Error analysis revealed that 1,000 instances were misclassified, with Classes 5, 6, and 3 showing the weakest performance, each with recall and F1-scores hovering around 0.88–0.89.

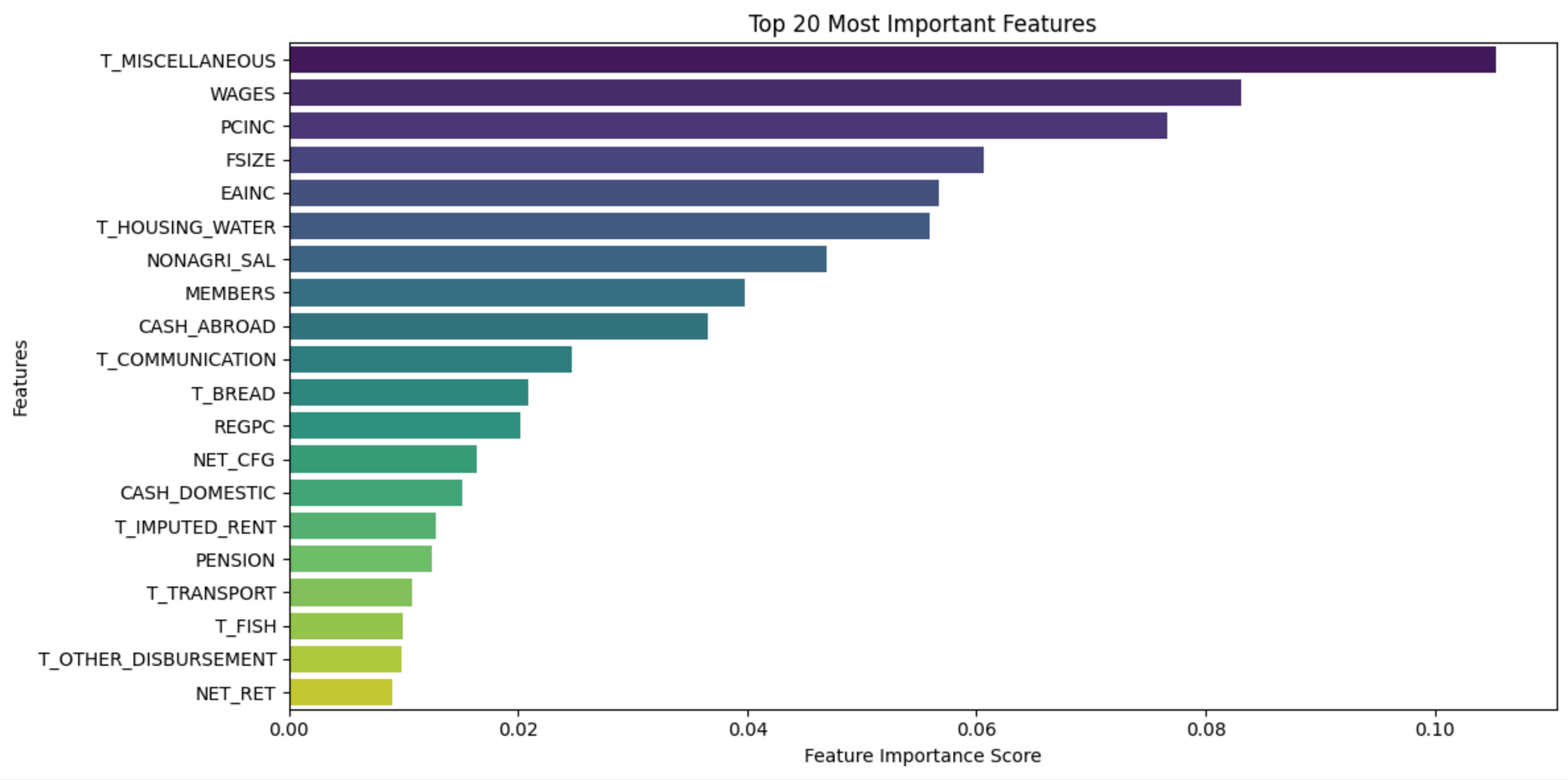
Misclassifications were most common between adjacent deciles, indicating difficulty in distinguishing neighboring income levels.

In contrast, Classes 0 and 9 achieved the highest accuracy (97.8% and 96.9%, respectively), likely reflecting more distinct feature distributions at the income extremes.

```
Classification Report:
              precision    recall  f1-score   support

           0       0.95      0.98      0.97      1344
           1       0.90      0.93      0.91      1355
           2       0.90      0.90      0.90      1374
           3       0.90      0.89      0.89      1268
           4       0.88      0.90      0.89      1240
           5       0.88      0.89      0.89      1145
           6       0.91      0.89      0.90      1107
           7       0.92      0.91      0.92      1093
           8       0.95      0.92      0.93      1066
           9       0.99      0.97      0.98      1060

    accuracy                           0.92     12052
   macro avg       0.92      0.92      0.92     12052
weighted avg       0.92      0.92      0.92     12052


Per-class Accuracy:
Class 0: 0.9784
Class 1: 0.9255
Class 2: 0.8981
Class 3: 0.8904
Class 4: 0.8992
Class 5: 0.8865
Class 6: 0.8871
Class 7: 0.9113
Class 8: 0.9231
Class 9: 0.9698

Number of misclassified samples: 1000
```

SECTION 3

XGBOOST - FEATURE IMPORTANCE



SECTION 3

XGBOOST - KEY INSIGHTS

XGBoost has a strong performance in predicting NATDC (National Income Deciles)

- ~92% testing accuracy after tuning
- Learns complex income-spending patterns
- However some overfitting observed (100% training accuracy)

Top Predictive Features

T_MISCELLANEOUS, WAGES, PCINC, FSIZE, EAINC, T_HOUSING_WATER

Key Players in Predicting

- Behavioral traits: Flexible spending (households in higher classes might spend more on hygienic products than those in the lower classes)
- Structural traits: Family size, income source (wages, pensions), remittances
- Geographic/economic context: Region-level income (REGPC), cash abroad

SECTION 3

XGBOOST - KEY INSIGHTS

Income Classification is Behavioral

- Higher deciles linked to lifestyle markers, not just earnings
- Larger families tend to be classified into lower deciles despite decent income

Diverse Income Sources Matter

- Stable wages (NONAGRI_SAL), pensions, and side income (NET_CFG, NET_RET) play distinct roles
- Entrepreneurial and informal income (EAINC) introduces variance across middle deciles

FIES Dataset Strengths

- Rich, granular mix of economic and demographic data
- Captures not just income, but how households live and spend

Challenges

- Overlap in middle deciles makes classification harder (3–6)
- Some inter-correlation and overfitting risk

SECTION 4

SUMMARY

The models kNN, Logistic Regression, XGBoost:

1. **k-Nearest Neighbors (kNN)**
    a. Performance: Accuracy improved from 33% to 37% (~4%) after tuning.
    b. Challenges:
        i. Struggles with high-dimension data, thus distance-based separability is weak
        ii. Strong bias toward Class 0, misclassifying mid-range classes (1–8).
2. **Logistic Regression**
    a. Performance: Accuracy reached 82% (initial training), but was outperformed by XGBoost.
3. **XGBoost**
    a. Performance: Achieved 92.2% accuracy, with a training accuracy of 100%, which could suggest mild overfitting.
    b. Challenges
        i. Misclassification due to overlapping features (Classes 3-6)
        ii. Intercorrelation between features and risk of overfitting

SECTION 4

SUMMARY

Model	Hyperparameters	Performance
KNN	Best parameters: {'metric': 'manhattan', 'n_neighbors': 1, 'weights': 'distance'}	Avg. Acc: 0.3727
Logistic Regression	Best parameters: model = LogisticRegression(solver="lbfgs", max_iter=1000, penalty="l2", C=50)	Avg. Acc: 81.57 (initial) Avg. Acc: 92.33 (C=50)
XGBoost	Best Parameters: {n_estimators=450, learning_rate=0.05819068217437232, max_depth=8, colsample_bytree=0.6567234139229252, subsample=0.9558111605361372, random_state=i }	Avg. Acc: 0.9224

SECTION 4

CONCLUSION

kNN

- kNN is not suitable for this dataset due to poor scalability and sensitivity to feature space complexities.

Logistic Regression

- A strong baseline model that provides reliable performance but lacks the complexity to capture nonlinear relationships.

XGBoost

- Best-performing model, effectively handling high-dimensional data and nonlinear patterns.

SECTION 4

CONCLUSION

Goal: Identify key determinants of household income decile (NATDC) and evaluate model performance.

Top Predictors:

- WAGES and CASH_ABROAD (Logistic Regression): Strong indicators of income level based on earnings.
- T_MISCELLANEOUS (XGBoost): Reflects household spending patterns—higher non-essential expenses linked to higher deciles.
- FSIZE and PCINC: Household structure and per capita income also play critical roles.

Model Performance Summary:

- XGBoost outperformed KNN and Logistic Regression with ~92% accuracy and stronger handling of complex feature interactions.
- Logistic Regression captured linear trends well.
- KNN underperformed in high-dimensional space, limited by distance-based assumptions.

SECTION 4

CONCLUSION

Weaknesses of what we did:

- Most of the features were selected mostly by logic given we wanted to know the National income decile
- We didn't scale the features to be balanced mostly due to how the data set is structured. We wanted to preserve it as much as possible; this gave rise to certain features having more importance than others due to how large their values are.
- As will be proven later due to how large wages is in terms of value it will predominantly dominate for most models which we believe to be fine as it is one of the key values we want for the label, NATDC
- As stated before we wanted to somewhat equalize the feature importance of each model to do this we added some of the expenses which based on our correlation matrix to be almost the same thus may lead to duplicates in the the model itself
- We also could have reduced the instances of dominating features such as those related income and expenditures so the model doesn't overly rely on them and ignore meaningful patterns in others.

SECTION 4

CONCLUSION

Final Insight:
Income classification is driven not just by earnings, but by how households spend and structure themselves. XGBoost captures this complexity best—making it a powerful tool for income-based classification and socio-economic analysis.

How can we improve?
Since most features were selected more on logic with minimal statistical tests we could improve feature selection by employing more statistical methods. This also brought the issue of the dataset not being scaled which might have been the cause of features (like wages) dominating the feature importance in certain models. While this dominance is acceptable given wages' importance we could have reduced the instances of the dominating features so that the model wouldn't over rely on them and acknowledge other features that may have brought more meaningful patterns

THANK YOU