Analyzing and Predicting Dwelling Occupancy in Washington State

Abstract:

The aim of this study is to predict whether a dwelling is occupied by owners or renters based on several features related to individual demographics and housing characteristics. We have used Support Vector Machines (SVM) to classify the dataset. The findings of this study provides a robust analysis of the factors influencing dwelling occupancy and uncover deeper patterns which will be useful to real estate professionals in understanding housing trends.

Introduction:

Renters tend to skew toward the lower ends of the economic scale when it comes to income and wealth, according to data from the Federal Reserve's 2019 Survey of Consumer Finances[1]. The primary goal of this study is to predict whether dwellings are occupied by owners or renters based on various demographic and housing-related factors. By using three different Support Vector Machines (SVM) kernels—linear, radial basis function (RBF), and polynomial, we not only seek to explore the predictive capabilities of these models but also gain insights into the underlying patterns within the data.

The dataset is obtained from the US Census, accessed through IPUMS USA[2]. This dataset is comprehensive with a wide range of variables including individual demographic information such as age, income, education level, and marital status as well as housing characteristics like electricity cost, year of construction, and population density of the surrounding area. These variables offer a rich source of information for understanding how individual attributes relate to whether people own or rent their homes.

Throughout this report, we will explore the dataset and pre-process it. Pre-processing the data plays an essential role in generating insights that we can trust. The goal is to understand the application of SVMs to classification tasks and analyze how different kernel functions influence model performance. We will examine the relationships between selected variables and housing occupancy.

Theoretical Background:

Support Vector Machines(SVMS) are supervised learning methods used for classification and regression problems. SVM is a common term used to refer to the maximal margin classifier, support vector machine and the support vector machine. However, Support vector machine is a generalization of maximal margin classifier which requires data to be linearly separable. The support vector classifier is an extension of the maximal margin classifier which can be applied to a broader variety of datasets. The support vector machine is an extension of the support vector classifier and SVM can be used in cases where the data has a non-linear boundary.

The support vector machine uses a hyperplane to separate the classes. A hyperplane in a p-dimensional space is a flat subspace of dimension p-1. In 2D, a hyperplane is a line and in 3D, it is a plane. When p>3, it's hard to visualize a hyperplane but the concept of p-1 dimensional subspace still applies. A hyperplane is in p-dimension is defined by the equation:

$$\beta 0 + \beta 1X1 + \beta 2X2 + \cdot \cdot \cdot + \beta pXp = 0 \qquad \text{eq(1)}$$

A point X= (X1,X2, . . . ,Xp)T in p-dimensional space lies on the hyperplane if it satisfies the eq(1). If eq(1) is greater than 0, X is on one side of the hyperplane, and if it is less than 0, X is on the other side. However, if data is perfectly linearly separable, there can be infinite hyperplanes as shown in figure 1 [3].

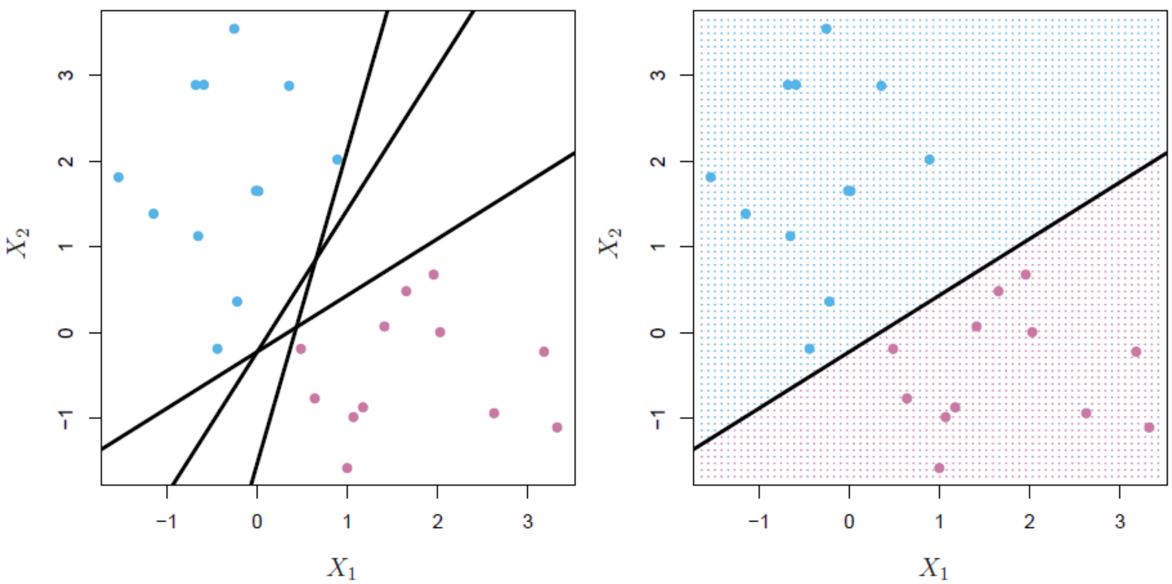


Figure 1

This is why choosing the hyperplane is crucial for the analysis, which is why we have the maximal margin classifier. It is the optimal hyperplane– separating two classes–that is farthest from the training data. The SVM uses the principle of maximizing the distance between nearest data points from the hyperplane from either class. The distance between the hyperplane and the point is called as margin as shown

The SVM operates on the idea of increasing the space between the closest points of each group and the dividing line, which is the hyperplane. The gap between this line and the nearest point is what we call the margin, as demonstrated in figure (2) [3].

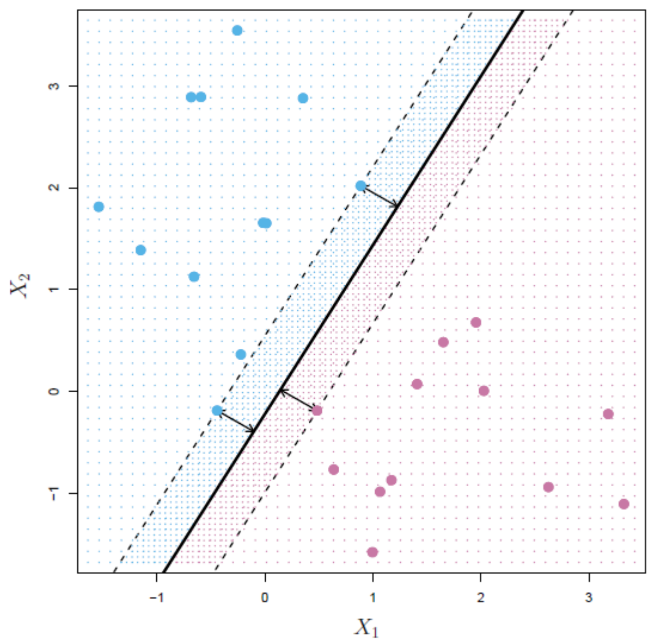


Figure 2

The nearest data points to the hyperplane are called the support vectors. Larger distance between the margin makes the model better. The mechanism of finding the hyperplane is fully accomplished by a subset of the training samples and the support vectors. Therefore, the support vectors play an essential role in determining the position of the hyperplane, and removing other training data points does not have any effect on the model but removing support vectors could affect our model performance drastically. They help in determining the optimal hyperplane for the dataset.

A classifier based on separating hyperplanes is good until an observation arrives which is far from the hyperplane, this shifts the hyperplane and results in a tiny margin. Then our confidence in the classification based on thin margin decreases. To tackle this, we have a Support vector classifier that is more robust to individual observations, although we make a trade-off here unlike in the Maximal Margin Classifier where we have a hard margin, here we allow few misclassifications and call it a soft-margin. We trade a small portion of our model's performance to get better overall results. Rather than having the optimal maximal margin so that data are on the correct side of the hyperplane and margin, we instead allow some data to be on the incorrect side of the margin, or even the hyperplane as shown in Figure (3) [3].

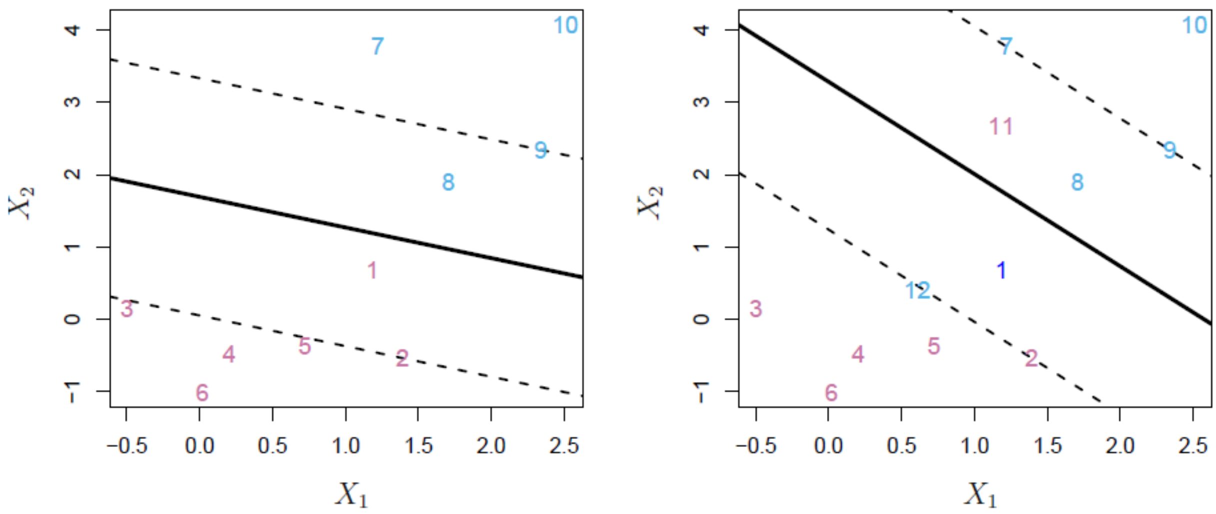


Figure 3: observations 11 and 12 are on the wrong side of the hyperplane and the wrong side of the margin.

This solution is defined as: $\boldsymbol{\epsilon_i \geq 0, \sum_{i=1}^{n} \epsilon_i \leq C}$ **eq(2)**

Where C is a tuning parameter. We try to find the highest possible C for our model, which means we want to allow misclassifications while also wanting our model to perform well.

If $\boldsymbol{\epsilon_i} = 0$ then the ith observation is on the correct side of the margin, If $\boldsymbol{\epsilon_i} > 0$ then the ith observation is on the wrong side of the margin, meaning it has violated the margin. If $\boldsymbol{\epsilon_i} > 1$ then it is on the wrong side of the hyperplane itself.

C tuning parameter is often referred to as the budget, because we have a certain budget to allow misclassifications. The value of c controls the tradeoff between the model's ability to fit the training data and its ability to generalize to new data. SVM's kernel hyperparameter is crucial for model performance, with options including linear, polynomial, and radial kernels. Different kernels enlarge the feature space in a specific way. Tuning these hyperparameters can have a significant effect on model accuracy.

Polynomial kernel is used when data is not linearly separable, so a polynomial function is used to separate the data. For instance, by increasing the feature space using quadratic, cubic, and even higher-order polynomial functions of the predictors.So instead of fitting a support vector classifier using p features $\boldsymbol{X_1, X_2, \ldots, X_p}$, we instead fit a support vector classifier using 2p features $\boldsymbol{X^1, X_1^2, X^2, X_2^2, \ldots, X_p, X_p^2}$. The basic idea is to transform the input data into a higher-dimensional space where it can be linearly separated, and then build a linear model on top of it to separate the classes. The kernel function used in polynomial kernel SVM is defined as

$$K(x_i, x_{i'}) = (1 + \sum_{j=1}^{p} x_{ij}x_{i'j})^d.$$

[3]

where x and y are input feature vectors, c is a constant, and d is the degree of the polynomial. When d is 1, the polynomial kernel is just a linear kernel, and when d is higher, the kernel function puts the data into a higher-dimensional space. The degree parameter controls the complexity of the polynomial function used to separate the data. If the degree is too low, the model may not be able to separate the data effectively, and if the degree is too high, the model might be overfitting and perform poorly on new data. Tuning these hyperparameters is important for achieving good performance with polynomial kernel SVMs.

The RBF kernel (Radial Basis Function), uses gamma to check if a new data point, let's say x^*, is near or far from the points we know. If x^* is far away, the radial kernel, with the help of gamma, basically says this point doesn't matter much for making predictions. This way, only the points that are really close to x^* influences what the prediction will be. RBF is defined as:

$$K(x_i, x_{i'}) = \exp(-\gamma \sum_{j=1}^{p} (x_{ij} - x_{i'j})^2).$$

Why do we not enlarge space using the original features? Using kernels is computationally lighter, because SVMs are computationally expensive and enlarging features could create infinite dimensions.

When there are more than 2 classes, the two most popular methods to use are the one-versus-one and one-versus-all. One-versus-one creates a classifier for every possible pair of classes. If we have K classes, this means we will have K(K−1)/2 classifiers. Each classifier votes for an observation to belong to one of two classes. The class that gets the most votes across all classifiers is the final choice for where the observation fits.

The one-versus-all method, also known as one-versus-rest, is another approach where we create one classifier per class, comparing each class against all the others combined. Each classifier gives a confidence score for its class, and the one with the highest score decides the class for the observation.

Methodology:

The dataset provided a unique challenge as each row represented a dwelling with multiple occupants living in the same house. SERIAL provides a unique identification number for each household in a dwelling and we can see that there are multiple records. More than 50% of the dataset is redundant.

checking for duplicate values

```
# @title checking for duplicate values
serial_duplicates = df.duplicated(subset=['SERIAL']).sum()
print(f"Number of duplicates based on SERIAL: {serial_duplicates}")
```

```
Number of duplicates based on SERIAL: 44586
```

I have subsetted the data by grouping each household by SERIAL column and taking the row which has the highest age, meaning the oldest individual of the household. This is because it is highly likely that they might be the renter or the owner.

Then we analyzed the dataset further and decided to remove these columns for the reasons mentioned below:

- **PERWT** variable is used when we want to do person analysis and it is not directly related to predicting home ownership, so having it is not required.
- **BRTHYR** can be deleted as we have the AGE variable, this is repetitive.
- **PERNUM** is not relevant since we are taking the eldest member from each family. But we will encode it like 1 for PERNUM > 1 and 0 for PERNUM ==1
- **AGE** We will take the maximum age from the rows as it is a good assumption that older individual in the household will be the owner or renter
- **EDUCD** : Will delete this column from the data set since it's correlated with the EDUC
- **INCTOT** : We will take mean for the INCTOT value
- **HHINCOME** : removing this as it's highly correlated to avg income INCTOT

There was a column VALUEH that holds the value of each household, this was removed from the dataset as it was making the model overfit, By analysing the Decision Tree, I was able to figure out that this column’s importance was =1 and it was a perfect separator. It makes sense because the prediction can be easily made by knowing the value of the household.

Svm is not good with too many categories, so we converted the MARST to a binary variable, 0 if an individual is single and 1 otherwise.

```
df_uni['MARST'].value_counts()
```

```
MARST
1    15780
6     5500
4     5429
5     2932
2      645
3      516
Name: count, dtype: int64
```

```
df_uni ['MARST'] = df_uni['MARST'].apply(lambda x: 0 if x == 6 else 1)
```

Then I addressed the data columns that included the cost of electricity, water, gas, and fuel. The predictors associated with these records had a particular code 9999, 9993, 9997 that indicated whether there was no cost or if these expenses were already included in the rent. It was crucial to replace these values with 0 to avoid model inaccuracy. After subsetting, i had 30k rows but I used 10K rows because the SVM was taking a long time to compute, 10k is also a good size for a dataste.

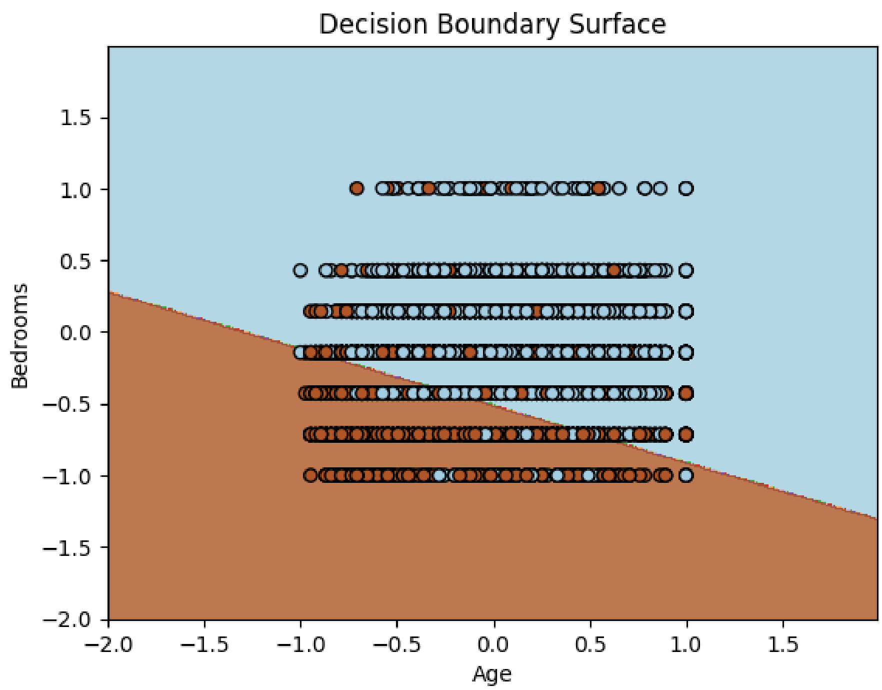
We used SVM for classification tasks to accurately identify the houses, using a variety of demographic and environmental parameters as predictors such as age, education level, family income, and cost of maintaining a property. I then used cross-validation techniques to assess the performance of our model. Later, I expanded the model to include the RBF kernel and polynomial kernel with varying cost, gamma, and polynomial degree values.

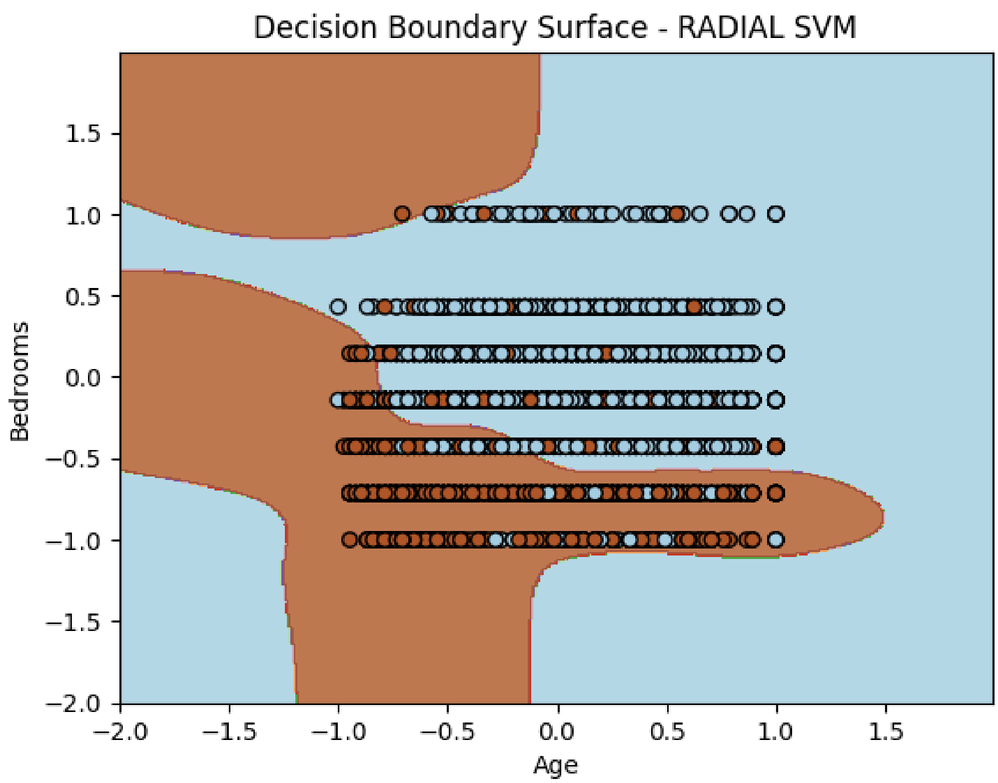
We found that even after training the model with such hyperparameters, the results were comparable to what we were achieving with the linear SVM model although Radial gave a slightly higher accuracy of 82%. Choosing the linear model as the final model is a good decision because it requires less computational effort to achieve the comparable outcomes. In summary, our process includes recognizing and addressing outliers, considering how to aggregate the data based on the unique features, and then training the SVM model and evaluating its performance on the test dataset.

Computational Results:

To build the model to classify the dwelling as rented or owned. I have used various variables like income, marital status, education and the cost which is involved in the household such as cost of gas, electricity and fuel. We have achieved 82% accuracy in classifying the dwelling. I have used linear, RBF and polynomial kernels to train the machine learning model. We cross validated the model using cost, gamma, and degree hyperparameters. It was observed that the model was giving almost similar accuracy with the RBF and linear kernel. Hence, it's advised to use the linear kernel since it is more robust and less prone to overfitting.

Plots for each model are shown below:





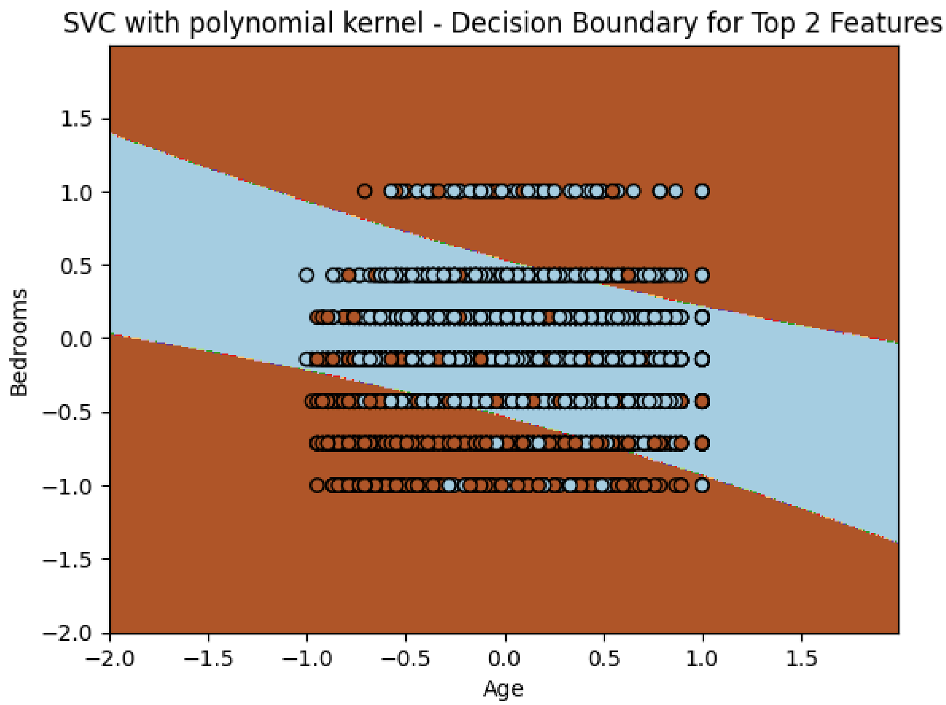
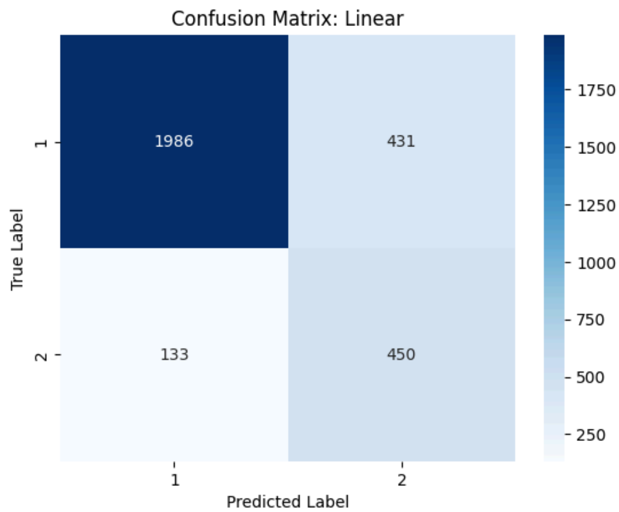


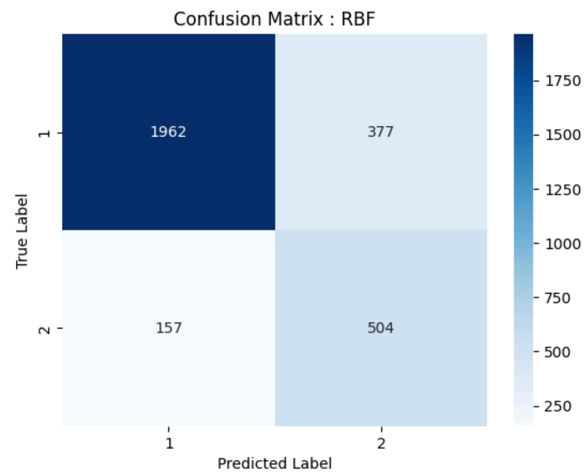
Figure 4

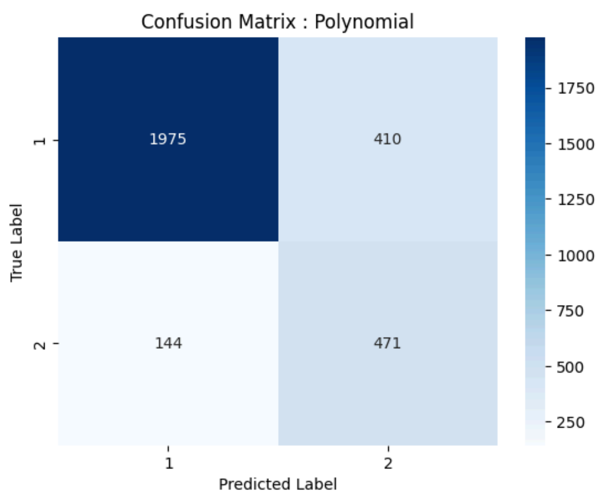
The model was working well on the test dataset0. I got almost the same accuracy and error on the test set as I was getting on the training set. It conveys that the model was not overfitting. We can see the decision boundary in Figure 4 for all the three models along with the confusion matrix on Figure 5 below and see how the model is performing. Since we can't plot the higher dimensions in 2D, we have plots the data using the Age and Rooms feature from the dataset. These two were the most important features. We can see that somehow the RBF kernel boundary might be overfitting the data.

	feature	importance
0	BEDROOMS	0.093133
1	AGE	0.036867
2	COSTWATR	0.009733
3	EDUC	0.006467
4	DENSITY	0.004800
5	COSTELEC	0.004200

Confusion matrix for all three models:







Discussion:

It is worth noting that the IPUMS USA dataset includes a vast array of data that can be used to answer many more questions related to the dwelling's ownership. This study concentrated specifically on the people's income, age, and education, which was just one component of the data. Regardless of this, our findings provide valuable insights into knowing which variables are important to determine if people will own the house or not. Our findings show that these factors have a considerable impact on a person's likelihood of purchasing a home later in life. The data could be further studied by answering questions based on parental education status and income and determining whether or not the person will buy their own home later in life. I also want to see how the results differ if we only look at the data for married couples. Apart from income and education, the study was based on several aspects such as the cost of maintaining the space such as electricity, fuel, and gas. According to the findings, Age, rooms, income and education play a crucial effect in deciding who will own the house. On test results, the SVM model attained an accuracy of 81%, suggesting its usefulness in identifying the residence. Furthermore, adding the RBF or polynomial kernel had no significant effect on the model's accuracy with RBF just doing 82%, slightly higher. However, the study had limitations such as the cross-sectional nature of the data, which limits the ability to infer causality, and self-reported data because the data was too correlated, resulting in data loss of nearly 50%. We didn't have many features to work with, such as bedrooms and rooms, which were significantly associated with each other. Other examples include the relationship between income and highest income, as well as education and education code. Furthermore, the sample size of the dataset was relatively small, which may limit the generalizability of findings to a larger population.

Conclusion:

This study focused on using Support Vector Machine (SVM) models and their kernel extensions, like RBF and polynomial, to predict ownership of a dwelling. We used data sourced from IPUMS USA, originally collected by the US Census. The results were promising, with the models reaching an accuracy of 82% (radial kernel) for the binary classification task of predicting home purchases versus rentals. This suggests that SVMs can be effective for real estate predictions. The strong performance of the models could be beneficial for real estate agents and policymakers in developing strategies that serve both companies and individuals. This study adds to the growing knowledge about applying SVMs with various kernels to home classification tasks.

References:

[1] Pew Research Center. (2022, March 23). Key facts about housing affordability in the U.S. Pew Research Center: Short Reads. https://www.pewresearch.org/short-reads/2022/03/23/key-facts-about-housing-affordability-in-the-u-s/

[2] Steven Ruggles, Sarah Flood, Matthew Sobek, Danika Brockman, Grace Cooper, Stephanie Richards, and Megan Schouweiler. IPUMS USA: Version 13.0 [dataset]. Minneapolis, MN: IPUMS, 2023. https://doi.org/10.18128/D010.V13.0

[3] James, G., Witten, D., Hastie, T., Tibshirani, R., & Taylor, J. (2023). An Introduction to Statistical Learning with Applications in Python. (Original work published 2023) https://hastie.su.domains/ISLP/ISLP_website.pdf.download.html

Appendix:

```
# !pip install ydata-profiling
```

```
import pandas as pd
import numpy as np
# from ydata_profiling import ProfileReport
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.metrics import roc_curve, accuracy_score
from mlxtend.plotting import plot_decision_regions
from sklearn.model_selection import train_test_split,GridSearchCV
from sklearn.linear_model import SGDClassifier
import matplotlib.pyplot as plt
from sklearn.preprocessing import MinMaxScaler
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
from sklearn.metrics import classification_report, confusion_matrix
%matplotlib inline
```

loading data

```
# @title loading data
df = pd.read_csv('Housing.csv')
df.shape,df.columns, df.dtypes
```

```
((75388, 24),
 Index(['SERIAL', 'DENSITY', 'OWNERSHP', 'OWNERSHPD', 'COSTELEC', 'COSTGAS',
        'COSTWATR', 'COSTFUEL', 'HHINCOME', 'VALUEH', 'ROOMS', 'BUILTYR2',
        'BEDROOMS', 'VEHICLES', 'NFAMS', 'NCOUPLES', 'PERNUM', 'PERWT', 'AGE',
        'MARST', 'BIRTHYR', 'EDUC', 'EDUCD', 'INCTOT'],
       dtype='object'),
 SERIAL         int64
 DENSITY      float64
 OWNERSHP       int64
 OWNERSHPD      int64
 COSTELEC       int64
 COSTGAS        int64
 COSTWATR       int64
 COSTFUEL       int64
 HHINCOME       int64
 VALUEH         int64
 ROOMS          int64
 BUILTYR2       int64
 BEDROOMS       int64
 VEHICLES       int64
 NFAMS          int64
 NCOUPLES       int64
 PERNUM         int64
 PERWT          int64
 AGE            int64
 MARST          int64
 BIRTHYR        int64
 EDUC           int64
 EDUCD          int64
 INCTOT         int64
 dtype: object)
```

Description of all varibles

Column	Description
SERIAL	Unique serial number of the record
DENSITY	Population density of the area
OWNERSHP	Ownership status (1 for owner, 2 for renter)
OWNERSHPD	Detailed ownership status
COSTELEC	Monthly electricity cost
COSTGAS	Monthly gas cost
COSTWATR	Monthly water cost
COSTFUEL	Monthly fuel cost
HHINCOME	Household income
VALUEH	House value
ROOMS	Number of rooms
BUILTYR2	Year of construction
BEDROOMS	Number of bedrooms
VEHICLES	Number of vehicles
NFAMS	Number of families in the household
NCOUPLES	Number of couples in the household

Column	Description
PERNUM	Person number within household
PERWT	indicates how many individuals in the U.S. population are statistically represented by a given person
AGE	Age of respondent
MARST	Marital status
BIRTHYR	Year of birth
EDUC	Education level
EDUCD	Detailed education level
INCTOT	Total pre-tax personal income or losses from all sources for the previous year.

```
df.head()
```

	SERIAL	DENSITY	OWNERSHP	OWNERSHPD	COSTELEC	COSTGAS	COSTWATR	COSTFUEL	HHIN(
0	1371772	920.0	1	13	9990	9993	360	9993	7!
1	1371773	3640.9	2	22	1080	9993	1800	9993	1:
2	1371773	3640.9	2	22	1080	9993	1800	9993	1:
3	1371774	22.5	1	13	600	9993	9993	9993	7
4	1371775	3710.4	2	22	3600	9993	9997	9993	5(

5 rows × 24 columns

checking for duplicate values

```
# @title checking for duplicate values
serial_duplicates = df.duplicated(subset=['SERIAL']).sum()
print(f"Number of duplicates based on SERIAL: {serial_duplicates}")
```

```
Number of duplicates based on SERIAL: 44586
```

missing values

```
# @title missing values
df.isnull().sum()
```

```
SERIAL       0
DENSITY      0
OWNERSHP     0
OWNERSHPD    0
COSTELEC     0
COSTGAS      0
COSTWATR     0
COSTFUEL     0
HHINCOME     0
VALUEH       0
ROOMS        0
BUILTYR2     0
BEDROOMS     0
VEHICLES     0
NFAMS        0
NCOUPLES     0
PERNUM       0
PERWT        0
AGE          0
MARST        0
BIRTHYR      0
EDUC         0
EDUCD        0
INCTOT       0
dtype: int64
```

```
# @title
# ProfileReport(df)
```

We have more than 50% of the dataset with duplicates based on SERIAL column which is unique for each household. We have 0 null values in dataset which is a good sign, our dataset is clean!

PERWT variable is used when we want to do person analysis and it is not directly related to predicting home ownership, so having it is not required.

BRTHYR can be deleted as we have the AGE variable, this is repetitive.

PERNUM is not relevant since we are taking the eldest member from each family. But we will encode it like 1 for PERNUM > 1 and 0 for PERNUM ==1

AGE: We will take the maximum age from the rows as it is a good assumption that older individual in the household will be the owner or renter

EDUCD : Will delete this column from the data set since it's correlated with the EDUC

INCTOT : We will take mean for the INCTOT value

HHINCOME : removing this as it's highly correlated to avg income

```
df['PERNUM'].value_counts()
```

```
PERNUM
1     30802
2     22732
3     10921
4      6339
5      2670
6      1099
7       454
8       196
9        89
10       48
11       20
12       10
13        3
14        3
15        1
16        1
Name: count, dtype: int64
```

```
df['PERNUM_CODED'] = df['PERNUM'].apply(lambda x: 0 if x == 1 else 1)
df['PERNUM_CODED'].head()
```

```
0    0
1    0
2    1
3    0
4    0
Name: PERNUM_CODED, dtype: int64
```

```
df['PERNUM_CODED'].value_counts()
```

```
PERNUM_CODED
1    44586
0    30802
Name: count, dtype: int64
```

```
df['INCTOT'] = df.groupby('SERIAL')['INCTOT'].transform('mean')
df.shape
```

```
(75388, 25)
```

Let's make the dataset unique by keeping rows with unique SERIAL and highest aged individual

```
df_uni = df.sort_values(by=['SERIAL', 'AGE'], ascending=[True, False]).drop_duplicates(subset='SERIAL')
df_uni.shape
```

```
(30802, 25)
```

```
df_uni.reset_index(inplace = True)
```

```
df_uni.shape
```

```
(30802, 26)
```

Dropping columns that are not important

```
# @title Dropping columns that are not important
df_uni = df_uni.drop(['PERWT', 'BIRTHYR','SERIAL', 'PERNUM', 'EDUCD', 'ROOMS', 'HHINCOME', 'VALUEH','OWNERSHPD','BUILTYR2'], axis=1)
```

```
df_uni.shape
```

```
(30802, 16)
```

```
df_uni[df_uni['AGE'] == 18].shape, df_uni[df_uni['AGE'] < 18].shape
```

```
((7, 16), (0, 16))
```

Our data doesn't have anyone aged less than 18, because younger people are not likely to own or rent a house

Let's make MARST a contunous variable

```
df_uni['MARST'].value_counts()
```

```
MARST
1    15780
6     5500
4     5429
5     2932
2      645
3      516
Name: count, dtype: int64
```

```
df_uni ['MARST'] = df_uni['MARST'].apply(lambda x: 0 if x == 6 else 1)
```

We can drop the original column now

```
# df_uni = df_uni.drop(['MARST'], axis=1)
# df_uni.shape
```

Analysing COST columns

```
# @title Analysing COST columns
df_uni['COSTWATR'].value_counts()
```

```
COSTWATR
9993    4533
9997    4078
1200    1825
1500     903
1000     871
        ...
620       20
690       19
3800      15
3900       9
3700       9
Name: count, Length: 133, dtype: int64
```

```
# @title
df_uni['COSTGAS'].value_counts()
```

```
COSTGAS
9993    15455
9992     3405
600      1209
360      1151
1200     1089
240       973
480       946
960       790
720       697
840       572
9997      519
120       472
1080      451
1800      431
2400      337
1440      315
1560      306
1320      262
7200      193
48        185
1680      167
3600      117
2160      112
1920      111
3000      111
2040      102
2280       54
2760       50
2640       40
2520       38
4200       34
2880       26
3360       19
3240       17
3480       14
3120       11
```

```
3720        7
4080        5
3840        5
3960        4
Name: count, dtype: int64
```

```
df_uni['COSTFUEL'].value_counts()
```

```
COSTFUEL
9993    27860
500       218
200       196
300       188
400       173
        ...
290         1
740         1
410         1
270         1
670         1
Name: count, Length: 109, dtype: int64
```

```
# @title
df_uni['COSTELEC'].value_counts()
```

```
1800    1906
2400    1737
600     1653
720     1579
1440    1493
840     1480
1080    1433
1560    1312
480     1133
1320    1046
1680     972
2160     889
9997     845
360      797
3000     745
1920     742
3600     725
2040     697
2280     536
9993     412
2760     403
2640     382
240      356
2520     304
2880     258
4200     243
4800     241
3360     220
9990     208
3120     172
3240     167
3480     130
6000     120
3960     112
120       93
3840      84
5400      77
3720      69
4080      68
4560      60
4440      47
48        45
4320      41
4680      33
5040      28
5280      21
5160      21
6600      21
5640      19
5760      17
4920      16
5520      14
6480       9
5880       9
6120       9
6240       7
6360       5
Name: count, dtype: int64
```

All columns have values like 9993,9992,9997 that has high value count, except for COSTELEC where with low value counts for these numbers. But it looks like these variables mean that the data is either missing or not correct, since the values are gonna affect the model, we'll recode these values to 0

Transforming COST columns

```
# @title Transforming COST columns

df_uni['COSTWATR'] = df_uni['COSTWATR'].apply(lambda x: 0 if x > 9992 else x)
df_uni['COSTELEC'] = df_uni['COSTELEC'].apply(lambda x: 0 if x > 9992 else x)
df_uni['COSTGAS'] = df_uni['COSTGAS'].apply(lambda x: 0 if x > 9992 else x)
df_uni['COSTFUEL'] = df_uni['COSTFUEL'].apply(lambda x: 0 if x > 9992 else x)
```

```
df_uni.shape, df_uni.dtypes
# df_uni['OWNERSHP'].value_counts()
```

```
((30802, 16),
 index             int64
 DENSITY         float64
 OWNERSHP          int64
 COSTELEC          int64
 COSTGAS           int64
 COSTWATR          int64
 COSTFUEL          int64
 BEDROOMS          int64
 VEHICLES          int64
 NFAMS             int64
 NCOUPLES          int64
 AGE               int64
 MARST             int64
 EDUC              int64
 INCTOT          float64
 PERNUM_CODED      int64
 dtype: object)
```

```
# df_uni = pd.get_dummies(df_uni, columns=['EDUC'], prefix='EDUC', dtype=int)
# df_uni.shape
```

```
df['EDUC'].value_counts()
```

```
EDUC
6     18937
10    14272
11     8932
7      8616
8      6103
1      5618
0      4622
2      4319
5      1463
4      1259
3      1247
Name: count, dtype: int64
```

```
def categorize_educ(x):
    if x == 0 or x == 99:
        return 0
    elif x > 0 and x <= 6:
        return 1
    elif x > 6 and x <= 11:
        return 2
    else:
        return 0
# Apply the categorize_educ function to create the EDUC category column
df['EDUC'] = df_uni['EDUC'].apply(categorize_educ)

# Check the value counts of the EDUC category column
print(df['EDUC'].value_counts())
```

```
EDUC
2.0    20347
1.0    10007
0.0      448
Name: count, dtype: int64
```

```
# df_uni['BUILTYR2'].value_counts()
```

```
df_uni.head()
```

	index	DENSITY	OWNERSHP	COSTELEC	COSTGAS	COSTWATR	COSTFUEL	BEDROOMS	VEHICLES
0	0	920.0	1	9990	0	360	0	4	2
1	1	3640.9	2	1080	0	1800	0	4	2
2	3	22.5	1	600	0	0	0	4	2
3	4	3710.4	2	3600	0	0	0	3	2
4	7	448.2	1	1560	3000	0	0	4	2

Next steps: Generate code with df_uni View recommended plots

Class imbalance, using stratify sampling

```
# @title Class imbalance, using stratify sampling
class_distribution = df_uni['OWNERSHP'].value_counts(normalize=True)

# Print the class distribution
print("Class Distribution:")
print(class_distribution)
```

```
Class Distribution:
OWNERSHP
1    0.707194
2    0.292806
Name: proportion, dtype: float64
```

```
sampled_df = df_uni.sample(n=10000, random_state=42)
sampled_df.shape
```

```
(10000, 16)
```

Splitting dataset and scaling the data

```
# @title Splitting dataset and scaling the data
from sklearn.preprocessing import StandardScaler
X = sampled_df.drop(['OWNERSHP'], axis=1)
y = sampled_df['OWNERSHP']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1, stratify=y)

scaling = MinMaxScaler(feature_range=(-1,1)).fit(X_train)
X_train = scaling.transform(X_train)
X_test = scaling.transform(X_test)
```

```
grid_params = {'C': [0.001,  0.1, 1,  10, 100]}
svcfit = SVC(kernel='linear',cache_size=1000,random_state = 1, max_iter = 1000)
tune = GridSearchCV(svcfit, grid_params ,cv=5)
tune.fit(X_train,y_train)
tune.best_params_,(1 - tune.best_score_) # # best C and cv error rate
```

```
({'C': 1}, 0.29800000000000004)
```

SVM is highly efficient but looks like the model is overfitting. Let's try Decision Tree Classififer and see if there's any column that's causing overfitting

Decision Tree Classifier

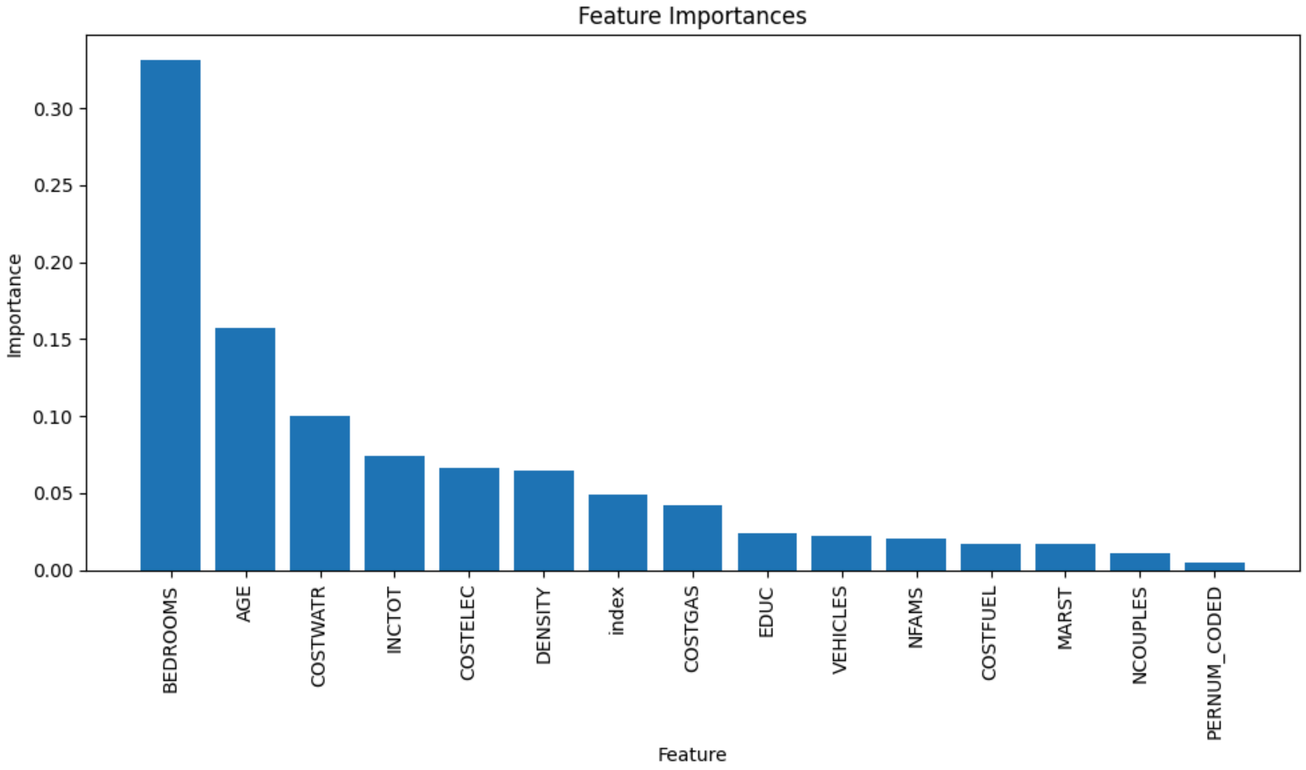
```
# @title Decision Tree Classifier
from sklearn.tree import DecisionTreeClassifier

clf = DecisionTreeClassifier(criterion='entropy', max_depth=10)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
accuracy_score(y_test, y_pred)
```

```
0.809
```

feature importances

```
# @title feature importances
importances = clf.feature_importances_
indices = np.argsort(importances)[::-1]  # Sort feature importances in descending order
print("Feature ranking:")
for f in range(X.shape[1]):
    print("%d. Feature '%s' (%f)" % (f + 1, X.columns[indices[f]], importances[indices[f]]))

# Plot the feature importances
plt.figure(figsize=(10, 6))
plt.title("Feature Importances")
plt.bar(range(X.shape[1]), importances[indices], align="center")
plt.xticks(range(X.shape[1]), X.columns[indices], rotation=90)
plt.xlabel("Feature")
plt.ylabel("Importance")
plt.tight_layout()
plt.show()
```

```
Feature ranking:
1. Feature 'BEDROOMS' (0.330914)
2. Feature 'AGE' (0.156908)
3. Feature 'COSTWATR' (0.099993)
4. Feature 'INCTOT' (0.074295)
5. Feature 'COSTELEC' (0.066339)
6. Feature 'DENSITY' (0.064927)
7. Feature 'index' (0.049052)
8. Feature 'COSTGAS' (0.042025)
9. Feature 'EDUC' (0.023538)
10. Feature 'VEHICLES' (0.022211)
11. Feature 'NFAMS' (0.020646)
12. Feature 'COSTFUEL' (0.017108)
13. Feature 'MARST' (0.016572)
14. Feature 'NCOUPLES' (0.010700)
15. Feature 'PERNUM_CODED' (0.004772)
```



VALUEH is the column that caused overfitting, let's not use it in our models.

Splitting dataset and scaling the data without VALUEH

```
# @title Splitting dataset and scaling the data without VALUEH
# df_uni = df_uni.drop(['VALUEH'], axis=1)

from sklearn.preprocessing import StandardScaler
X = sampled_df.drop(['OWNERSHP'], axis=1)
y = sampled_df['OWNERSHP']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1, stratify=y)

scaling = MinMaxScaler(feature_range=(-1,1)).fit(X_train)
X_train = scaling.transform(X_train)
X_test = scaling.transform(X_test)


grid_params = {'C': [0.001,  0.1, 1,  10, 100]}
svcfit = SVC(kernel='linear',cache_size=10000,random_state = 1, max_iter = 20000)
tune = GridSearchCV(svcfit, grid_params ,cv=5)
tune.fit(X_train,y_train)
print('Best Parameters: ',tune.best_params_)
print('Lowest cross-validation error rate: {:.2f}%'.format(1 - tune.best_score_))
```

```
Best Parameters:  {'C': 10}
Lowest cross-validation error rate: 0.16%
```

```
# Assuming you have already defined grid_params, X_train, y_train as per your snippet and run the GridSearchCV

# Retrieve the best estimator
best_svc = tune.best_estimator_

# Get the coefficients from the best model
coefficients = best_svc.coef_[0]

# Rank the features by the absolute value of their coefficients
feature_importance = np.abs(coefficients)

# Get the indices of the features, sorted by importance
sorted_indices = np.argsort(feature_importance)[::-1]

# Print out the ranked features
print("Feature ranking:")
for idx in sorted_indices:
    print(f"Feature {idx}, Coefficient: {coefficients[idx]}")
```

```
Feature ranking:
Feature 6, Coefficient: -2.4401393666857345
Feature 8, Coefficient: 2.0009077892301086
Feature 4, Coefficient: -1.0978772825282306
Feature 10, Coefficient: -1.0410002718281355
Feature 2, Coefficient: -0.8580817070340387
Feature 5, Coefficient: -0.8271943675704705
Feature 9, Coefficient: -0.5583389225579722
Feature 12, Coefficient: -0.5482499402379695
Feature 1, Coefficient: 0.3810677043161732
Feature 13, Coefficient: 0.2736896558892461
Feature 3, Coefficient: -0.17556460433500476
Feature 7, Coefficient: 0.13570037140660318
Feature 11, Coefficient: -0.08646238430658215
Feature 0, Coefficient: 0.05084770255204507
Feature 14, Coefficient: 0.016710572324523554
```

```
y_pred = tune.predict(X_test)
print(classification_report(y_test,y_pred))
```

```
              precision    recall  f1-score   support

           1       0.85      0.92      0.89      2119
           2       0.77      0.62      0.69       881

    accuracy                           0.83      3000
   macro avg       0.81      0.77      0.79      3000
weighted avg       0.83      0.83      0.83      3000
```

Top 5 features

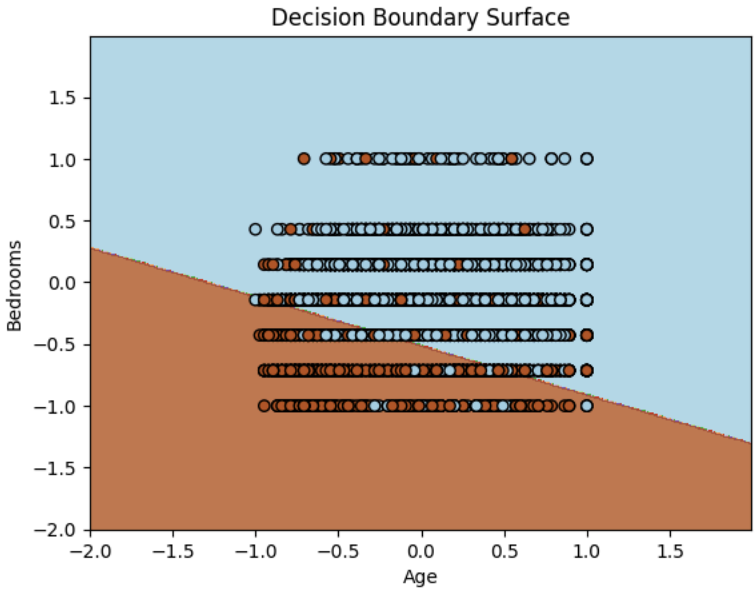
BEDROOMS , NFAMS, COSTWATR, AGE, COSTELEC

```
#sorting top features using permuration importance

from sklearn.feature_selection import SelectKBest, f_classif
from sklearn.inspection import permutation_importance

k = 15
selector = SelectKBest(f_classif, k=k)
selector.fit(X_train, y_train)
X_train_new = selector.transform(X_train)
X_test_new = selector.transform(X_test)

svc_linear = SVC(kernel='linear', C=0.1, cache_size=10000, random_state=1, max_iter=20000)
svc_linear.fit(X_train, y_train)

result = permutation_importance(svc_linear, X_test_new, y_test, n_repeats=5, random_state=1)
importance_df = pd.DataFrame({'feature': X.columns[selector.get_support()], 'importance': result.importances_mean})
importance_df = importance_df.sort_values(by='importance', ascending=False).reset_index(drop=True)
importance_df
```

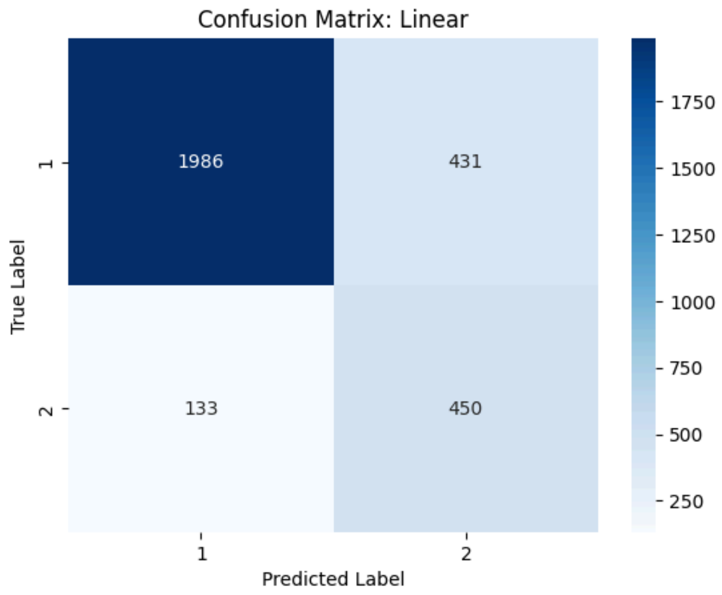
	feature	importance
0	BEDROOMS	0.093133
1	AGE	0.036867
2	COSTWATR	0.009733
3	EDUC	0.006467
4	DENSITY	0.004800
5	COSTELEC	0.004200
6	INCTOT	0.002933
7	NFAMS	0.001933
8	COSTFUEL	0.001600
9	VEHICLES	0.001200
10	NCOUPLES	0.000533
11	COSTGAS	0.000333
12	index	-0.000600
13	PERNUM_CODED	-0.000867
14	MARST	-0.001400

Next steps: Generate code with `importance_df` View recommended plots

```
# Update the model to use the best C parameter and fit it to the top two features from your training set
imp_df = sampled_df[[ 'AGE', 'BEDROOMS','OWNERSHP']]

X = imp_df.drop(['OWNERSHP'], axis=1)
y = imp_df['OWNERSHP']

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=1, stratify=y)

scaling = MinMaxScaler(feature_range=(-1,1)).fit(X_train)
X_train = scaling.transform(X_train)
X_test = scaling.transform(X_test)

svc_ = SVC(kernel='linear', C=0.1, cache_size=10000, random_state=1, max_iter=20000)
svc_.fit(X_train, y_train)  # Update to use features 6 and 8

class_labels = ['Class 0', 'Class 1']
# define a grid of points
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
Z = svc_.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# plot the decision boundary surface
# plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)

plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Paired, edgecolors='k')
plt.xlabel('Age')
plt.ylabel('Bedrooms')
# plt.legend(class_labels)
plt.title('Decision Boundary Surface')
plt.show()
```




```
y_pred = svc_.predict(X_test)

# create confusion matrix
conf_matrix = pd.crosstab(index=y_pred, columns=y_test, rownames=[''])

# create a heatmap of the confusion matrix
sns.heatmap(conf_matrix, annot=True, cmap="Blues", fmt="d")

# add labels to the plot
plt.title("Confusion Matrix: Linear")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```




```
print('Test Error',np.mean(y_pred != y_test))
print('Test Accuracy',np.mean(y_pred == y_test))
```

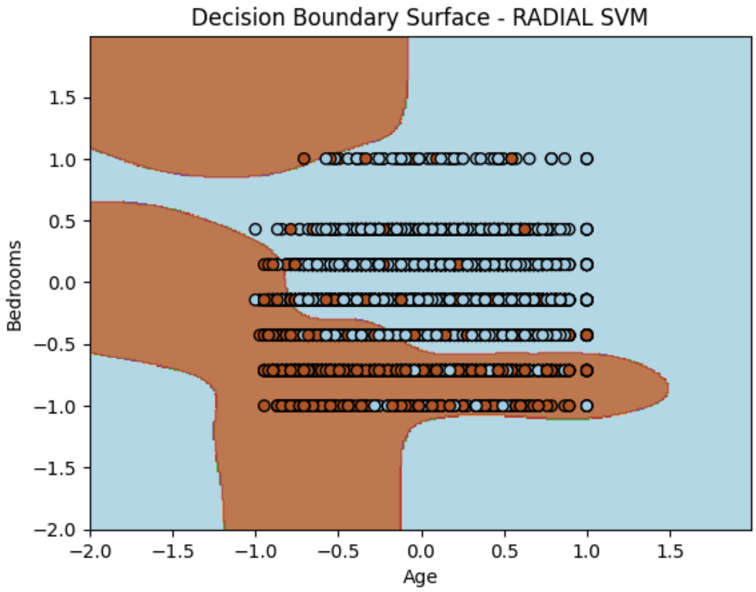
```
Test Error 0.188
Test Accuracy 0.812
```

Radial Kernel

```
# @title Radial Kernel
grid_params = {'C': [0.1,1,10,100],'gamma' : [2,4]}
svcfit_rbf = SVC(kernel = 'rbf',cache_size = 10000,random_state = 42, max_iter = 15000)
tune_rbf = GridSearchCV(svcfit_rbf, grid_params ,cv=10)
tune_rbf.fit(X_train,y_train)

print('COST: ',tune_rbf.best_params_)
print('Best Cross-validation error rate: {:.2f}%'.format(1 - tune_rbf.best_score_))
print('classification_report of on test data')
y_pred_rbf = tune_rbf.predict(X_test)
print(classification_report(y_test,y_pred))
```

```
COST:  {'C': 100, 'gamma': 2}
Best Cross-validation error rate: 0.18%
classification_report of on test data
              precision    recall  f1-score   support

           1       0.82      0.94      0.88      2119
           2       0.77      0.51      0.61       881

    accuracy                           0.81      3000
   macro avg       0.80      0.72      0.75      3000
weighted avg       0.81      0.81      0.80      3000
```

```
svc_best = SVC(kernel='rbf', C=tune_rbf.best_params_['C'],gamma = tune_rbf.best_params_['gamma'],cache_size=10000, random_state=42, max_

# fit the classifier to the training data
svc_best.fit(X_train, y_train)  # Update to use features 6 and 8

class_labels = ['Class 0', 'Class 1']
# define a grid of points
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
Z = svc_best.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# plot the decision boundary surface
# plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.contourf(xx, yy, Z, cmap=plt.cm.Paired, alpha=0.8)

plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Paired, edgecolors='k')
plt.xlabel('Age')
plt.ylabel('Bedrooms')
# plt.legend(class_labels)
plt.title('Decision Boundary Surface - RADIAL SVM')
plt.show()
```




```
y_pred = tune_rbf.predict(X_test)

# create confusion matrix
conf_matrix = pd.crosstab(index=y_pred, columns=y_test, rownames=[''])

# create a heatmap of the confusion matrix
sns.heatmap(conf_matrix, annot=True, cmap="Blues", fmt="d")

# add labels to the plot
plt.title("Confusion Matrix : RBF")
plt.xlabel("Predicted Label")
plt.ylabel("True Label")
plt.show()
```


```
# @title
print('Test Error',np.mean(y_pred != y_test))
print('Test Accuracy',np.mean(y_pred == y_test))
```

```
Test Error 0.178
Test Accuracy 0.822
```

Polynomial

```
# @title Polynomial
svcfit_poly = SVC(kernel = 'poly',cache_size = 10000,max_iter=15000, random_state = 1)
grid_params = {'C': [0.1,1,10],'gamma' : [2,4], 'degree' : [2,3,4]}
tune_poly = GridSearchCV(svcfit, grid_params ,cv=10)
tune_poly.fit(X_train,y_train)
print('COST: ',tune_poly.best_params_)
print('Least CV error rate: {:.2f}%'.format(1 - tune_poly.best_score_))
y_pred = tune_poly.predict(X_test)
print(classification_report(y_test,y_pred))
```

```
COST:  {'C': 10, 'degree': 2, 'gamma': 2}
Least CV error rate: 0.19%
              precision    recall  f1-score   support

           1       0.83      0.93      0.88      2119
           2       0.77      0.53      0.63       881

    accuracy                           0.82      3000
   macro avg       0.80      0.73      0.75      3000
weighted avg       0.81      0.82      0.80      3000
```

```
# @title
svc_best_poly = SVC(kernel='poly', C=tune_poly.best_params_['C'],gamma = tune_poly.best_params_['gamma'],degree = tune_poly.best_params_
svc_best_poly.fit(X_train, y_train)

class_labels = ['Class 0', 'Class 1']

# define a grid of points
x_min, x_max = X_train[:, 0].min() - 1, X_train[:, 0].max() + 1
y_min, y_max = X_train[:, 1].min() - 1, X_train[:, 1].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.01), np.arange(y_min, y_max, 0.01))
Z = svc_best_poly.predict(np.c_[xx.ravel(), yy.ravel()])
Z = Z.reshape(xx.shape)

# plot the decision boundary surface
# plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)
plt.contourf(xx, yy, Z, cmap=plt.cm.Paired)

plt.scatter(X_train[:, 0], X_train[:, 1], c=y_train, cmap=plt.cm.Paired, edgecolors='k')
plt.xlabel('Age')
plt.ylabel('Bedrooms')  # Update the label to the actual feature name if known
plt.title('SVC with polynomial kernel - Decision Boundary for Top 2 Features')
plt.show()
```