Predicting The Cost of a One-Bedroom Apartment in Major World Cities

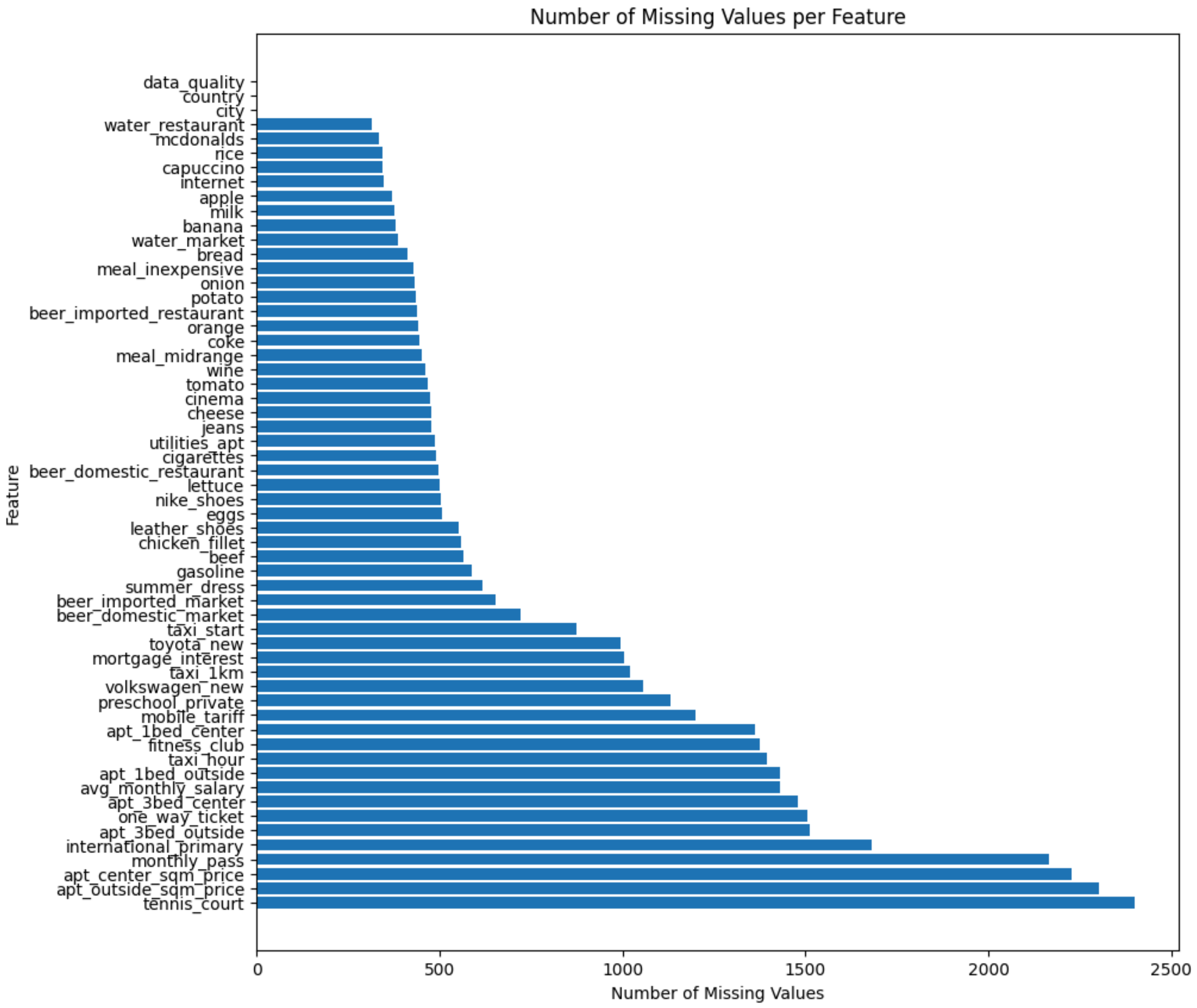
First, let's read in our dataset. This was obtained from: https://www.kaggle.com/datasets/mvieira101/global-cost-of-living?resource=download

[ ] ↳ 2 cells hidden

Data Preparation and Cleaning

First, let's examine the overall features and missing values in our dataset.

```
missing = data.isnull().sum()
missing_sorted = missing.sort_values(ascending=False)
plt.rcParams['figure.figsize'] = [10, 10]
plt.title('Number of Missing Values per Feature')
plt.barh(missing_sorted.index, missing_sorted.values)
plt.ylabel('Feature')
plt.xlabel('Number of Missing Values')
plt.show()
```



We are trying to predict the "**apt_1bed_center**" feature, which is the price of a one-bedroom apartment in the city center. So we can drop the features that are too closely correlated to our target feature--namely, the price of a 3 bedroom apartment inside our outside the city center, the price of a 1 bedroom apartment outside the city center, and the average price per square meter of apartments inside or outside the city center. These are all too similar to what we're predicting.

Dropping these values is also good because they happen to be missing a lot of data in the original dataset. There are other features such as **"international_primary", "monthly_pass", and "tennis_court"** that also have high volumes of missing data that we might want to look out for when testing models, as if they aren't important features than dropping them could provide us with more data to work with.

The dataset contains a "data quality" column that identifies which records are considered to have good data quality by the creator. However, only keeping those rows for training yields just 744 records for training and testing, which is 15% of the original dataset size.

Thus, to have more data, we will focus on first dropping the unnecessary columns and then removing records with missing values, without worrying about data quality.

We can drop any rows with a missing value for any one of the features to clean up our data. This leaves us with 1427 records in the dataset to use for training/testing.

```
# # For later use
# data_good_qual = data[data['data_quality'] != 0]
# Clean data
data_clean = data.drop(['apt_1bed_outside', 'apt_3bed_outside', 'apt_3bed_center',
                  'apt_center_sqm_price', 'apt_outside_sqm_price'], axis=1)
data_clean.dropna(inplace=True)
data_clean.drop_duplicates(inplace=True)
data_clean
```

	city	country	meal_inexpensive	meal_midrange	mcdonalds	beer_domestic_restaurant	beer_imported_restaurant	capuccino	cok
0	Seoul	South Korea	7.68	53.78	6.15	3.07	4.99	3.93	1.4
1	Shanghai	China	5.69	39.86	5.69	1.14	4.27	3.98	0.5
2	Guangzhou	China	4.13	28.47	4.98	0.85	1.71	3.54	0.4
3	Mumbai	India	3.68	18.42	3.68	2.46	4.30	2.48	0.4
4	Delhi	India	4.91	22.11	4.30	1.84	3.68	1.77	0.4
...	...	...	...	...	...	...	...	...	.
4826	Lysa nad Labem	Czech Republic	4.33	32.40	5.19	1.25	2.29	2.15	1.2
4910	Livigno	Italy	14.23	52.69	7.38	3.42	4.21	1.50	2.5
4928	Murovani Kurylivtsi	Ukraine	2.72	12.24	2.69	0.68	1.09	0.68	0.2
4945	Tirupati	India	2.46	9.21	4.30	2.21	3.07	1.47	0.4
4950	Egilsstadhir	Iceland	17.01	70.87	8.50	4.25	3.54	3.90	1.7

1427 rows × 53 columns

Next, we can look at which features are most closely correlated with the target variable (price of a 1 bed apartment in the city center)?

```
plt.figure(figsize=(10, 30))
plt.subplots_adjust(hspace=0.2)
# plt.suptitle("Correlations")

for i in range(3):
  cur_method = 'pearson'
  if i == 1:
    cur_method = 'spearman'
  elif i == 2:
    cur_method = 'kendall'
  corr = data_clean.corr(method=cur_method).abs()
  corr_y = corr['apt_1bed_center'].sort_values(ascending=False)

  ax = plt.subplot(3,1,i+1)
  ax.barh(corr_y.index, corr_y.values)
  ax.set_xlabel('Correlation Values')
  ax.set_ylabel('Features')
  ax.set_title('Correlation with Price of Apartment (' + cur_method + ')')
```

Data Exploration

↳ *4 cells hidden*

Anomaly Detection (Data Cleaning + Exploration)

While we've removed the missing values, there could be outliers in the data and removing them would be helpful for testing our models. We can clearly see from the plots above that there are some outliers that have extremely high values for the apartment price variable, which should probably be removed. We run anomaly detection with the isolation forest algorithm to find and specifically identify outliers like this. This algorithm is good for our purposes because it is fast, not susceptible to curse of dimensionality (we have many features in our dataset), and it does not require significant tuning of hyperparameters (only the anomaly threshold value to identify an anomaly).

[ ] ↳ *11 cells hidden*

Data Analysis: Classification

Naive Bayes Model

In the Naive Bayes mode, we computed a log transformation on label values to apply a normaly distribution to the y values. Most of the y-values fall within a certain range that was encompassed mostly within 1 bin despite variations in total bin sizes. transform y into log y since y is not a normal distribution.It is recommended there is atleast 10 members per bin but we had two instances of bins with only 3 and 1 members. When we tested the naive bayes model by binning the y values (one-bedroom apartment prices in the city) into 3, 5 and 10 bins and performing cross validation, binning the labels into 3 values perf0rmed better with an average score of 70.5% compared to labels binned into 5 and 10 bins with average accuracies of 65.7% and 50%. This is indicates that it is harder to predict which price range a one-bedroom partment will be classified as when the range is small.

```
from sklearn.naive_bayes import GaussianNB
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import KBinsDiscretizer


#DO SMOTE STUFF POSSIBLY*
X = data_clean.drop(['apt_1bed_center'], axis=1)
y = data_clean['apt_1bed_center']
y_transformed = np.log(y)
y_transformed = y_transformed.to_numpy()

# Reshape y_array into a column vector
y_array_reshaped = y_transformed.reshape(-1, 1)

# Create a discretizer object with 3, 5 and 10 bins
discretizer = KBinsDiscretizer(n_bins=3, encode='ordinal', strategy='uniform')
discretizer2 = KBinsDiscretizer(n_bins=5, encode='ordinal', strategy='uniform')
discretizer3 = KBinsDiscretizer(n_bins=10, encode='ordinal', strategy='uniform')
```

```
# Fit and transform the target variable y
y_discrete = discretizer.fit_transform(y_array_reshaped)
y_discrete2 = discretizer2.fit_transform(y_array_reshaped)
y_discrete3 = discretizer3.fit_transform(y_array_reshaped)

# Convert the output back to a 1D array
y_discrete = y_discrete.ravel()
y_discrete2 = y_discrete2.ravel()
y_discrete3 = y_discrete3.ravel()


# Create naives bayes model
naive_bayes = GaussianNB()
features = X

#Cross Validation
scores = cross_val_score(naive_bayes,X=features,y=y_discrete, cv = 10)
scores2 = cross_val_score(naive_bayes,X=features,y=y_discrete2, cv = 10)
scores3 = cross_val_score(naive_bayes,X=features,y=y_discrete3, cv = 10)

print("Accuracy with 3 bins:", scores.mean())
print("Accuracy with 5 bins:", scores2.mean())
print("Accuracy with 10 bins:", scores3.mean())
```

It seems evident that the highest accuracy is one where you have to predict with the least number of bins. We fit the naive bayes classifiers to the various bins and print utilize the predict function to make a classification report.

```
from sklearn.model_selection import cross_val_score, cross_val_predict
from sklearn.metrics import confusion_matrix, classification_report

naive_bayes_3 = naive_bayes.fit(X = features, y = y_discrete)
naive_bayes_5 = naive_bayes.fit(X = features, y = y_discrete2)
naive_bayes_10 = naive_bayes.fit(X = features, y = y_discrete3)

naive_bayes_predict = cross_val_predict(naive_bayes,X=features,y=y_discrete, cv = 10)
print("Confusion Matrix with 3 bins:")
print(confusion_matrix(y_discrete, naive_bayes_predict))
print("Classification Report with 3 bins:")
print(classification_report(y_discrete, naive_bayes_predict))

naive_bayes_predict2 = cross_val_predict(naive_bayes,X=features,y=y_discrete2, cv = 10)
print("Confusion Matrix with 5 bins:")
print(confusion_matrix(y_discrete2, naive_bayes_predict2))
print("Classification Report with 5 bins:")
print(classification_report(y_discrete2, naive_bayes_predict2))

naive_bayes_predict3 = cross_val_predict(naive_bayes,X=features,y=y_discrete3, cv = 10)
print("Confusion Matrix with 10 bins:")
print(confusion_matrix(y_discrete2, naive_bayes_predict3))
print("Classification Report with 10 bins:")
print(classification_report(y_discrete2, naive_bayes_predict3))
```

The roc_curve function does not support multiclass classification. Since there are more than two classes/bins we cannot use roc_curve directly.One possible solution is to use the One-vs-Rest (OvR) strategy to convert the multiclass problem into several binary classification problems. In OvR, for each class, a binary classifier is trained to distinguish samples belonging to that class from the rest of the samples. Then, the outputs of these binary classifiers can be used to compute the ROC curve and AUC for each class.

```
from sklearn.metrics import roc_curve, roc_auc_score
from sklearn.multiclass import OneVsRestClassifier
from sklearn.model_selection import train_test_split

#separate test and training data
test_train3 = train_test_split(features, y_discrete, test_size=.2, train_size=.8, random_state = 33)
test_train5 = train_test_split(features, y_discrete2, test_size=.2, train_size=.8, random_state = 33)
test_train10 = train_test_split(features, y_discrete3, test_size=.2, train_size=.8, random_state = 33)

# create a One-vs-Rest classifier with GaussianNB as the base classifier
ovr_nb3 = OneVsRestClassifier(GaussianNB())
ovr_nb5 = OneVsRestClassifier(GaussianNB())
ovr_nb10 = OneVsRestClassifier(GaussianNB())
# fit the One-vs-Rest classifier on the training data
ovr_nb3.fit(test_train3[0], test_train3[2])
ovr_nb5.fit(test_train5[0], test_train5[2])
ovr_nb10.fit(test_train10[0], test_train10[2])

# predict the probabilities for each class using the One-vs-Rest classifier
probabilities3 = ovr_nb3.predict_proba(test_train3[1])
probabilities5 = ovr_nb5.predict_proba(test_train5[1])
probabilities10 = ovr_nb10.predict_proba(test_train10[1])

# compute the ROC curve and AUC for each class
fpr3 = dict()
tpr3 = dict()
auc3 = dict()
for i in range(len(ovr_nb3.classes_)):
    fpr3[i], tpr3[i], _ = roc_curve(test_train3[3], probabilities3[:, i], pos_label=i)
    auc3[i] = roc_auc_score(test_train3[3] == i, probabilities3[:, i])

fpr5 = dict()
tpr5 = dict()
auc5 = dict()
for i in range(len(ovr_nb5.classes_)):
    fpr5[i], tpr5[i], _ = roc_curve(test_train5[3], probabilities5[:, i], pos_label=i)
    auc5[i] = roc_auc_score(test_train5[3] == i, probabilities5[:, i])

fpr10 = dict()
tpr10 = dict()
auc10 = dict()
for i in range(len(ovr_nb10.classes_)):
    fpr10[i], tpr10[i], _ = roc_curve(test_train10[3], probabilities10[:, i], pos_label=i)
    auc10[i] = roc_auc_score(test_train10[3] == i, probabilities10[:, i])

for i in range(len(ovr_nb3.classes_)):
    plt.plot(fpr3[i], tpr3[i], label=f'class {i} (AUC = {auc3[i]:.2f})')

plt.plot([0,1],[0,1],'k--')
plt.xlabel('fpr')
plt.ylabel('tpr')
plt.title('ROC Curve Naive Bayes with 3 Bins')
plt.legend(loc='lower right')
plt.show()

for i in range(len(ovr_nb5.classes_)):
    plt.plot(fpr5[i], tpr5[i], label=f'class {i} (AUC = {auc5[i]:.2f})')

plt.plot([0,1],[0,1],'k--')
plt.xlabel('fpr')
plt.ylabel('tpr')
plt.title('ROC Curve Naive Bayes with 5 Bins')
plt.legend(loc='lower right')
plt.show()

for i in range(len(ovr_nb10.classes_)):
    plt.plot(fpr10[i], tpr10[i], label=f'class {i} (AUC = {auc10[i]:.2f})')

plt.plot([0,1],[0,1],'k--')
plt.xlabel('fpr')
plt.ylabel('tpr')
plt.title('ROC Curve Naive Bayes with 10 Bins')
plt.legend(loc='lower right')
plt.show()
```

Random Over Sampler

Instead of manually transforming the data to make up for class imbalance, let's utilize the ROS package and bin the range into 5 groups. This duplicates minority class records. With ROS, we see a 10% increase in accuracy of the classifier and through cross validation as opposed to manually transforming the data.

```
from sklearn.model_selection import train_test_split
from imblearn.over_sampling import SMOTE
from sklearn.naive_bayes import GaussianNB
from sklearn.preprocessing import KBinsDiscretizer
from imblearn.over_sampling import RandomOverSampler

# Instantiate and fit Random Over Sampler object
ros = RandomOverSampler()
y_whole = np.round(y).astype(int)
X_resampled, y_resampled = ros.fit_resample(X, y_whole)

y_whole = y_resampled.to_numpy().reshape(-1,1)
# Discretize both the training and testing sets
discretizer = KBinsDiscretizer(n_bins=5, encode='ordinal', strategy='uniform')
discretizer_ytrain = discretizer.fit_transform(y_whole)
discretizer_ytrain = discretizer_ytrain.ravel()

naive_bayes0 = GaussianNB()
features = X_resampled
scores_0 = cross_val_score(naive_bayes0,X=features,y=discretizer_ytrain, cv = 10)
print("Cross Validation Accuracy with 5 bins:", scores_0.mean())
```

```
from numpy.lib.histograms import histogram
print(np.unique(test_train3[2]))
print(np.unique(y_discrete3, return_counts= True))
plt.hist(np.log(y))
```

K Nearest Neighbors

The knn pipeline is built using the StandardScaler and PCA classes from sklearn.preprocessing and the KNeighborsClassifier class from sklearn.neighbors. The code then defines a parameter grid using the param_grid dictionary that contains the hyperparameters for PCA and KNeighborsClassifier. It specifies the number of components to keep after PCA and the number of neighbors to use for k-NN. Afterwards, a grid search is utilized to find the best hyperparameters for the pipeline. The following cell involves knn with binning of 5 and ROS data. With the optimal number of bins and ROS data, the best number of neighbors is 1 with an accuracy of 98% however this is likely due to overfitting.

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.neighbors import KNeighborsClassifier
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

scaler = StandardScaler()
pca = PCA(svd_solver = 'full')
knn = KNeighborsClassifier()
pipeline = Pipeline(steps=[('scaler', scaler), ('pca', pca), ('knn', knn)])

param_grid = {
    'pca__n_components': [0.7, 0.8, 0.9, 0.95],
    'knn__n_neighbors': list(range(1, 25))
}

grid_search5 = GridSearchCV(pipeline, param_grid, cv=5)
grid_search5.fit(features, discretizer_ytrain)

print("Best number of dimensions with 5 bins:", grid_search5.best_params_['pca__n_components'])
print("Best number of neighbors with 5 bins:", grid_search5.best_params_['knn__n_neighbors'])
print("Best accuracy with 5 bins:", grid_search5.best_score_)
```

```
Best number of dimensions with 5 bins: 0.9
Best number of neighbors with 5 bins: 1
Best accuracy with 5 bins: 0.9814964610717898
```

When we perform knn for 3, 5 and 10 bins without ROS, we get significantly lower accuracy scores with an almost 30% decrease.

```
scores = cross_val_score(pipeline, X, y_discrete, cv=5)
print("Accuracy for 3 discrete bins:  ", scores.mean())

scores2 = cross_val_score(pipeline, X, y_discrete2, cv=5)
print("Accuracy for 5 discrete bins:  ", scores2.mean())

scores3 = cross_val_score(pipeline, X, y_discrete3, cv=5)
print("Accuracy for 10 discrete bins:  ", scores3.mean())
```

KNN Without Binning

We had to bin in the Naive Bayes model but let's try KNN without binning. When we do a knn regressor with ROS, we don't get accuracies as high as knn with binning wih the best number of beighbors = 7 and an accuracy of 59%

```
from sklearn.neighbors import KNeighborsRegressor
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled, test_size=0.2, random_state=42)

for n in range(1, 11):
    knn = KNeighborsRegressor(n_neighbors=n)
    knn.fit(X_train, y_train)
    scores = cross_val_score(knn, X, y, cv=5)
    print("KNN without binning with", n, "neightbors:", scores.mean())
```

Support Vector Classifier

Since we have already created bins for our data, lets try a support vector machine. We use the KBinsDiscretizer from earlier.

```
from sklearn.preprocessing import KBinsDiscretizer
from sklearn.svm import SVC
from sklearn.model_selection import cross_val_score

# Create naives bayes model
svc = SVC()
features = X

#Cross Validation
scores = cross_val_score(svc, X=features, y=y_discrete, cv = 10)
scores2 = cross_val_score(svc, X=features, y=y_discrete2, cv = 10)
scores3 = cross_val_score(svc,X=features,y=y_discrete3, cv = 10)

print("Accuracy with 3 bins:", scores.mean())
print("Accuracy with 5 bins:", scores2.mean())
print("Accuracy with 10 bins:", scores3.mean())
```

```
/usr/local/lib/python3.9/dist-packages/sklearn/model_selection/_split.py:700: UserWarning: The least populated class in y has only 3 mem
  warnings.warn(
Accuracy with 3 bins: 0.7727000299670364
Accuracy with 5 bins: 0.6034212366396965
Accuracy with 10 bins: 0.35635800619318747
```

Once again, we find that using a smaller amount of bins results in a more accurate classifier.

Logistic Regression

First, the code calculates the median rent of one-bedroom apartments and creates a new column in the data_clean dataframe called above_median_rent, which is 1 if the rent is above the median and 0 otherwise.

Next, the data is split into training and testing sets using train_test_split() from scikit-learn. The training set contains 80% of the data and the remaining 20% is used for testing.

A logistic regression model is trained using LogisticRegression() from scikit-learn, with the training data as input. The model is then used to predict the target variable for the test data using predict(). The accuracy of the model is then calculated using accuracy_score() from scikit-learn.

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

median_rent = data_clean['apt_1bed_center'].median()
data_clean['above_median_rent'] = (data_clean['apt_1bed_center'] > median_rent).astype(int)

# Split the data into training and testing sets
X_reg = data_clean.drop(['above_median_rent'], axis=1)
y_reg = data_clean['above_median_rent']
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(X_reg, y_reg, test_size=0.2, random_state=42)

# Train a logistic regression model
clf = LogisticRegression()
clf.fit(X_train_reg, y_train_reg)

# Evaluate the model on the testing set
y_pred_reg = clf.predict(X_test_reg)
accuracy = accuracy_score(y_test_reg, y_pred_reg)
print(f"Accuracy: {accuracy}")
```

The accuracy is too high for Logistic Regression. An accuracy that is too high could be a sign of overfitting. Overfitting occurs when a model is too complex and fits the training data too closely, leading to poor performance on unseen data. In the case of a logistic regression model, an accuracy that is too high could mean that the model is relying too much on the training data and is not generalizing well to new data. This can be problematic because the model may not perform well on real-world data that it was not trained on.

```
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.preprocessing import StandardScaler

median_rent = data_clean['apt_1bed_center'].median()
data_clean['above_median_rent'] = (data_clean['apt_1bed_center'] > median_rent).astype(int)

X_reg = data_clean.drop(['above_median_rent'], axis=1)
y_reg = data_clean['above_median_rent']
X_train_reg, X_test_reg, y_train_reg, y_test_reg = train_test_split(X_reg, y_reg, test_size=0.2, random_state=42)

scaler = StandardScaler()
X_train_reg = scaler.fit_transform(X_train_reg)
X_test_reg = scaler.transform(X_test_reg)

clf = LogisticRegression(max_iter=1000)
scores = cross_val_score(clf, X_train_reg, y_train_reg, cv=5)
print(f"Cross validation scores: {scores}")
print(f"Mean accuracy: {scores.mean()}")

clf.fit(X_train_reg, y_train_reg)
y_pred_reg = clf.predict(X_test_reg)
accuracy = accuracy_score(y_test_reg, y_pred_reg)
print(f"Accuracy on testing data: {accuracy}")
```

```
Cross validation scores: [0.96035242 0.97356828 0.9339207  0.96460177 0.96460177]
Mean accuracy: 0.9594089899029278
Accuracy on testing data: 0.9507042253521126
```

```
import numpy as np
from sklearn.model_selection import cross_val_score

# Define the logistic regression model
clf = LogisticRegression(C=1.0, max_iter=1000, solver='lbfgs')

# Perform k-fold cross-validation
cv_scores_train = cross_val_score(clf, X_train_reg, y_train_reg, cv=5, scoring='accuracy')
cv_scores_test = cross_val_score(clf, X_test_reg, y_test_reg, cv=5, scoring='accuracy')

# Calculate the average train and test scores
train_score = np.mean(cv_scores_train)
test_score = np.mean(cv_scores_test)

# Print the scores
print(f"Average train score: {train_score}")
print(f"Average test score: {test_score}")
```

```
     Average train score: 0.9594089899029278
     Average test score: 0.9190476190476191
```

```
from sklearn.model_selection import cross_val_score
from sklearn.preprocessing import KBinsDiscretizer


X = data_clean.drop(['apt_1bed_center'], axis=1)
y = data_clean['apt_1bed_center']


lr = LogisticRegression()
features = X

#Cross Validation
scores = cross_val_score(lr,X=features,y=y_discrete, cv = 10)
scores2 = cross_val_score(lr,X=features,y=y_discrete2, cv = 10)
scores3 = cross_val_score(lr,X=features,y=y_discrete3, cv = 10)

print("Accuracy with 3 bins:", scores.mean())
print("Accuracy with 5 bins:", scores2.mean())
print("Accuracy with 10 bins:", scores3.mean())
```

Decision Tree Classification

We can use decision trees for classification by binning the continuous target variable into several bins and treating those as class labels.

```
from sklearn.preprocessing import KBinsDiscretizer

X = data_clean.drop(['apt_1bed_center'], axis=1)
y = data_clean['apt_1bed_center']
```

Let's test each of these discretizations with ensemble decision tree methods. We will display results for Random Forest, since after testing we realized that each of Random Forest, Gradient Boosting, and Bagging produced similar results.

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import confusion_matrix, classification_report, accuracy_score

model = RandomForestClassifier()
print('Random Forest Classifier Results:')
preds = cross_val_predict(model,X,y_discrete,cv=5)
print('Accuracy with 3 bins:')
print(accuracy_score(y_discrete, preds))
print("Confusion Matrix with 3 bins:")
print(confusion_matrix(y_discrete, preds))
print("Classification Report with 3 bins:")
print(classification_report(y_discrete, preds))

preds = cross_val_predict(model,X,y_discrete2,cv=5)
print('Accuracy with 5 bins:')
print(accuracy_score(y_discrete2, preds))
print("Confusion Matrix with 5 bins:")
print(confusion_matrix(y_discrete2, preds))
print("Classification Report with 5 bins:")
print(classification_report(y_discrete2, preds))

preds = cross_val_predict(model,X,y_discrete3,cv=5)
print('Accuracy with 10 bins:')
print(accuracy_score(y_discrete3, preds))
print("Confusion Matrix with 10 bins:")
print(confusion_matrix(y_discrete3, preds))
print("Classification Report with 10 bins:")
print(classification_report(y_discrete3, preds))
```

These classification methods have good accuracies, the best accuracies coming from the discretizations where the bins are the largest. This makes sense since there are less classes to identify.

However, as we increase the number of bins, the accuracy gets worse, and we can't predict the price of a 1 bed apartment in more "fine" bins. Also, since we have to label each of the bins with a class, this uses label encoding, which might cause the model to capture some numerical relationship among the class labels.

Let's try regression models to see if we can predict the exact prices with similar accuracy, using the R-squared metric.

Data Analysis: Regression Models

Linear Regression

The code imports the necessary libraries, including pandas for data manipulation and Linear Regression, train_test_split and RFE for feature selection. The data is split into training and testing sets, and the RFE (Recursive Feature Elimination) method is used to select the top five features that are most relevant to the target variable (rent). The RFE method works by recursively removing the least important feature and fitting the model again until the desired number of features is reached.

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.feature_selection import RFE

X_ln = data_clean.drop(['apt_1bed_center', 'above_median_rent'], axis=1)
y_ln = data_clean['apt_1bed_center']
X_train_ln, X_test_ln, y_train_ln, y_test_ln = train_test_split(X_ln, y_ln, test_size=0.2, random_state=42)

model = LinearRegression()

rfe = RFE(model, n_features_to_select=5)
rfe.fit(X_train_ln, y_train_ln)

print("Feature ranking:", rfe.ranking_)

ranking = rfe.ranking_


feature_ranking = {}
for i in range(len(X_ln.columns)):
    feature_ranking[X_ln.columns[i]] = ranking[i]

sorted_features = sorted(feature_ranking.items(), key=lambda x: x[1])
print("sorted features ", sorted_features)

for feature, rank in sorted_features:
    print(f"{feature}: {rank}")
```

The code keeps track of the highest R-squared value and the best set of features found so far. The best_features list is updated with the best set of features found in each iteration, and the highest_r_squared variable is updated with the highest R-squared value found.

Overall, this code is an example of a systematic approach to feature selection using the RFE method and a Linear Regression model. The resulting set of features can then be used to build a more accurate predictive model.

```
X_train_ln, X_test_ln, y_train_ln, y_test_ln = train_test_split(X_ln, y_ln, test_size=0.2, random_state=42)

highest_r_squared = 0.0
best_features = []

while len(X_train_ln.columns) > 4:
    model = LinearRegression()

    rfe = RFE(model, n_features_to_select=5)
    rfe.fit(X_train_ln, y_train_ln)

    print("Feature ranking:", rfe.ranking_)

    ranking = rfe.ranking_

    feature_ranking = {}
    for i in range(len(X_train_ln.columns)):
        feature_ranking[X_train_ln.columns[i]] = ranking[i]

    sorted_features = sorted(feature_ranking.items(), key=lambda x: x[1])

    best_features_for_iteration = [sorted_features[i][0] for i in range(5)]

    model.fit(X_train_ln[best_features_for_iteration], y_train_ln)
    r_squared = model.score(X_test_ln[best_features_for_iteration], y_test_ln)

    if r_squared > highest_r_squared:
        highest_r_squared = r_squared
        best_features = best_features_for_iteration

    print("Best features for iteration:", best_features_for_iteration)
    print("R-squared for iteration:", r_squared)
    print("Current best features:", best_features)
    print("Current highest R-squared:", highest_r_squared)
    print("")

    X_train_ln = X_train_ln.drop(best_features_for_iteration, axis=1)
    X_test_ln = X_test_ln.drop(best_features_for_iteration, axis=1)
```

```
from sklearn.model_selection import KFold
from sklearn.linear_model import LinearRegression
from sklearn.metrics import r2_score
import numpy as np

scores = cross_val_score(LinearRegression(), X, y, cv=5)
print(scores)
print('Generalization Estimate R2 Score: ' + str(scores.mean()))

# k = 5
# kf = KFold(n_splits=k, shuffle=True, random_state=42)

# X_ln = data_clean.drop('apt_1bed_center', axis=1)
# y_ln = data_clean['apt_1bed_center']

# r2_scores = []

# for train_index, test_index in kf.split(X_ln):

#     X_train_ln, X_test_ln = X_ln.iloc[train_index], X_ln.iloc[test_index]
#     y_train_ln, y_test_ln = y_ln.iloc[train_index], y_ln.iloc[test_index]

#     model = LinearRegression()
#     model.fit(X_train_ln, y_train_ln)

#     y_pred_ln = model.predict(X_test_ln)
#     r2 = r2_score(y_test_ln, y_pred_ln)
#     print(" R-squared value: ", r2)

#     r2_scores.append(r2)

# print("Average R-squared value:", np.mean(r2_scores))
```

```
     [0.85731802 0.84946468 0.82878574 0.81961978 0.74574502]
     Generalization Estimate R2 Score: 0.8201866485597314
```

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import GridSearchCV
import numpy as np

scaler = StandardScaler()
lin_reg = LinearRegression()


pipeline = Pipeline(steps=[('scaler', scaler), ('lin_reg', lin_reg)])

param_grid = {
    'lin_reg__fit_intercept': [True, False],
    'lin_reg__copy_X': [True, False]
}

grid_search = GridSearchCV(pipeline, param_grid, cv=5, scoring='r2')
grid_search.fit(X_ln, y_ln)

print("Best hyperparameters:", grid_search.best_params_)
print("Best R2 Score:", grid_search.best_score_)
```

Support Vector Regression

To build the svr model, we used a pipeline using the StandardScaler and PCA classes again. Using a parameter grid that adjusts the hyperparameters for PCA and SVR to find the optimal values. We then use cross_val_predict to perform nested cross validation.

```
from sklearn.svm import SVR
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import cross_val_predict


scaler = StandardScaler()
pca = PCA(svd_solver='full')
svr = SVR()
pipeline = Pipeline(steps=[('scaler', scaler), ('pca', pca), ('svr', svr)])
param_grid = {
    'pca__n_components': [0.6, 0.7, 0.8, 0.9, 0.95],
    'svr__kernel': ['linear', 'rbf', 'poly']
}

oned_y = np.ravel(y)
grid_search = GridSearchCV(pipeline, param_grid, cv=5, error_score='raise')
grid_search.fit(X, oned_y)
print("Best number of dimensions: ", grid_search.best_params_['pca__n_components'])
print("Best kernel type: ", grid_search.best_params_['svr__kernel'])
pred = cross_val_predict(grid_search, X, oned_y, cv=5)
r2 = r2_score(y, pred)
print("R-Squared: ", r2)
```

```
     Best number of dimensions:  0.95
     Best kernel type:  linear
     R-Squared:  0.8008451105089884
```

Using the optimal hyperparameters we found above, lets visualize our model by looking at the SVR with a few different features.

```
xscaler = StandardScaler()
yscaler = StandardScaler()
cap = data_clean[['capuccino']]
cap = xscaler.fit_transform(cap)
y = yscaler.fit_transform(y.to_numpy().reshape(-1,1))
svr = SVR(kernel='linear')
svr.fit(cap, y)

plt.scatter(cap, y, color = 'red')
plt.plot(cap, svr.predict(cap), color = 'blue')
plt.title('Support Vector Regression Model')
plt.xlabel('Price of a capuccino (scaled)')
plt.ylabel('Price of a 1 bedroom apartment (scaled)')
plt.show()

toyota = data_clean[['toyota_new']]
toyota = xscaler.fit_transform(toyota)
svr.fit(toyota, y)

plt.scatter(toyota, y, color = 'red')
plt.plot(toyota, svr.predict(toyota), color = 'blue')
plt.title('Support Vector Regression Model')
plt.xlabel('Price of a new toyota (scaled)')
plt.ylabel('Price of a 1 bedroom apartment (scaled)')
plt.show()

eggs = data_clean[['eggs']]
eggs = xscaler.fit_transform(eggs)
svr.fit(eggs, y)

plt.scatter(eggs, y, color = 'red')
plt.plot(eggs, svr.predict(eggs), color = 'blue')
plt.title('Support Vector Regression Model')
plt.xlabel('Price of eggs (scaled)')
plt.ylabel('Price of a 1 bedroom apartment (scaled)')
plt.show()
```

Decision Trees for Regression

We will test decision tree regressors, specifically using the ensemble methods Random Forest, Gradient Boosting, and Bagging. This will hopefully boost the performance of the decision trees by aggregating the predictions of multiple weak learners.

```
def get_adj_r2(r2, n, p):
    return (1-(1-r2)*((n-1)/(n-p-1)))
```

```
from sklearn.model_selection import train_test_split, cross_val_score
from sklearn.ensemble import RandomForestRegressor, GradientBoostingRegressor, BaggingRegressor
from sklearn.metrics import mean_squared_error, r2_score
```

For this particular train-test split, we can examine to see which features contributed the most to each model using mean decrease in impurity.

```
y = data_clean['apt_1bed_center']

plt.figure(figsize=(8,16))
for i in range(3):
  if i == 0:
    model = rf
    model_name = 'Random Forest'
  elif i == 1:
    model = gb
    model_name = 'Gradient Boosting'
  else:
    model = bg
    model_name = 'Bagging'
  if model_name == 'Bagging':
    importances = np.mean([
      tree.feature_importances_ for tree in model.estimators_
    ], axis=0)
  else:
    importances = model.feature_importances_
  forest_importances = pd.Series(importances,
                                 index=X_train.columns).sort_values()
  ax = plt.subplot(3,1,i+1)
  forest_importances.plot.barh(ax=ax)
  ax.set_title(model_name + " Feature importances (MDI)")
  ax.set_ylabel("Mean Decrease In Impurity")

plt.tight_layout()
model = bg.fit(X_train, y_train)
```

The average monthly salary of the city seems to dominate in these predictions based on the MDI metric. We will try removing some features to see if the models as a whole rely solely on this metric, or if it is just a strong predictor of 1 bedroom apartment price.

```
r2 = r2_score(y_test, preds)
```

Let's use 5-fold cross validation so our results are validated across all the training records.

```
rf = RandomForestRegressor()
gb = GradientBoostingRegressor()
bg = BaggingRegressor()
scores = cross_val_score(rf, X, y, cv=5)
print('Random Forest CV R2 Score:', scores.mean())
scores = cross_val_score(rf, X, y, cv=5)
print('Gradient Boosting CV R2 Score:', scores.mean())
scores = cross_val_score(rf, X, y, cv=5)
print('Bagging CV R2 Score:', scores.mean())
```

```
     Random Forest CV R2 Score: 0.8419983308769485
     Gradient Boosting CV R2 Score: 0.8415948781808174
     Bagging CV R2 Score: 0.8419955570255107
```

Decision Trees: Feature Selection

We can try reducing the dimensionality of our dataset using PCA.

```
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.pipeline import Pipeline
from sklearn.model_selection import GridSearchCV

param_grid = {
    'pca__n_components': [0.7, 0.8, 0.9, 0.95]
}
scaler = StandardScaler()
pca = PCA(svd_solver='full')
rf = RandomForestRegressor()
pipeline = Pipeline([('scaler', scaler), ('pca', pca), ('rf', rf)])
grid_search = GridSearchCV(pipeline, param_grid, cv=5)
grid_search.fit(X, y)
```