Part A

1. Explain below the 5 components shown in orange boxes. Explain which Azure components you will use where in this big data architecture and why.

Azure Data Lake: used for Ingest Data, Data Store, Prepare and transform data

A data lake is a centralized repository that ingests and stores large volumes of data in its original form. The data can then be processed and used as a basis for a variety of analytic needs. Due to its open, scalable architecture, a data lake can accommodate all types of data from any source, from structured (database tables, Excel sheets) to semi-structured (XML files, webpages) to unstructured (images, audio files, tweets), all without sacrificing fidelity. The data files are typically stored in staged zones—raw, cleansed, and curated—so that different types of users may use the data in its various forms to meet their needs. Data lakes provide core data consistency across a variety of applications, powering big data analytics, machine learning, predictive analytics, and other forms of intelligent action.

Azure Databricks: used for Ingest Data, Data Store, Prepare and transform data, Model and serve data

Azure Databricks is used to process, store, clean, share, analyze, model, and monetize their datasets with solutions from BI to machine learning. You can use the Azure Databricks platform to build many different applications spanning data personas. Customers who fully embrace the lakehouse take advantage of our unified platform to build and deploy data engineering workflows, machine learning models, and analytics dashboards that power innovations and insights across an organization.

Azure Data Factory: used for Ingest Data, Data Store, Prepare and transform data

It is the cloud-based ETL and data integration service that allows you to create data-driven workflows for orchestrating data movement and transforming data at scale. Using Azure Data Factory, you can create and schedule data-driven workflows (called pipelines) that can ingest data from disparate data stores. You can build complex ETL processes that transform data visually with data flows or by using compute services such as Azure HDInsight Hadoop, Azure Databricks, and Azure SQL Database.

Azure Synapse Analytics: used for Ingest Data, Prepare and transform data, Model and serve data

Azure Synapse Analytics is a scalable and cloud-based data warehousing solution from Microsoft. It is the next iteration of the Azure SQL data warehouse. It provides a unified environment by combining the data warehouse of SQL, the big data analytics capabilities of Spark, and data integration technologies to ease the movement of data between both, and from external data sources. We can ingest, prepare, manage, and serve data for immediate BI and machine learning needs easily with Azure Synapse Analytics.

Azure Cosmos DB: used for Ingest Data, Data Store, Prepare and transform data, Model and serve data

Cosmos Database (DB) is a horizontally scalable, globally distributed, fully managed, low latency, multi-model, multi query-API database for managing data at large scale. Cosmos DB is a PaaS (Platform as a Service) offering from Microsoft Azure and is a cloud-based NoSQL database. Cosmos DB is sometimes referred to as a serverless database, and it is a highly available, highly reliable, and high throughput database. Cosmos DB is a superset of Azure Document DB and is available in all Azure regions. With Cosmos DB you can distribute the data to any number of Azure regions, i.e., the data can be replicated to the geolocation from where your users are accessing, which helps in serving data quickly to users with low latency.

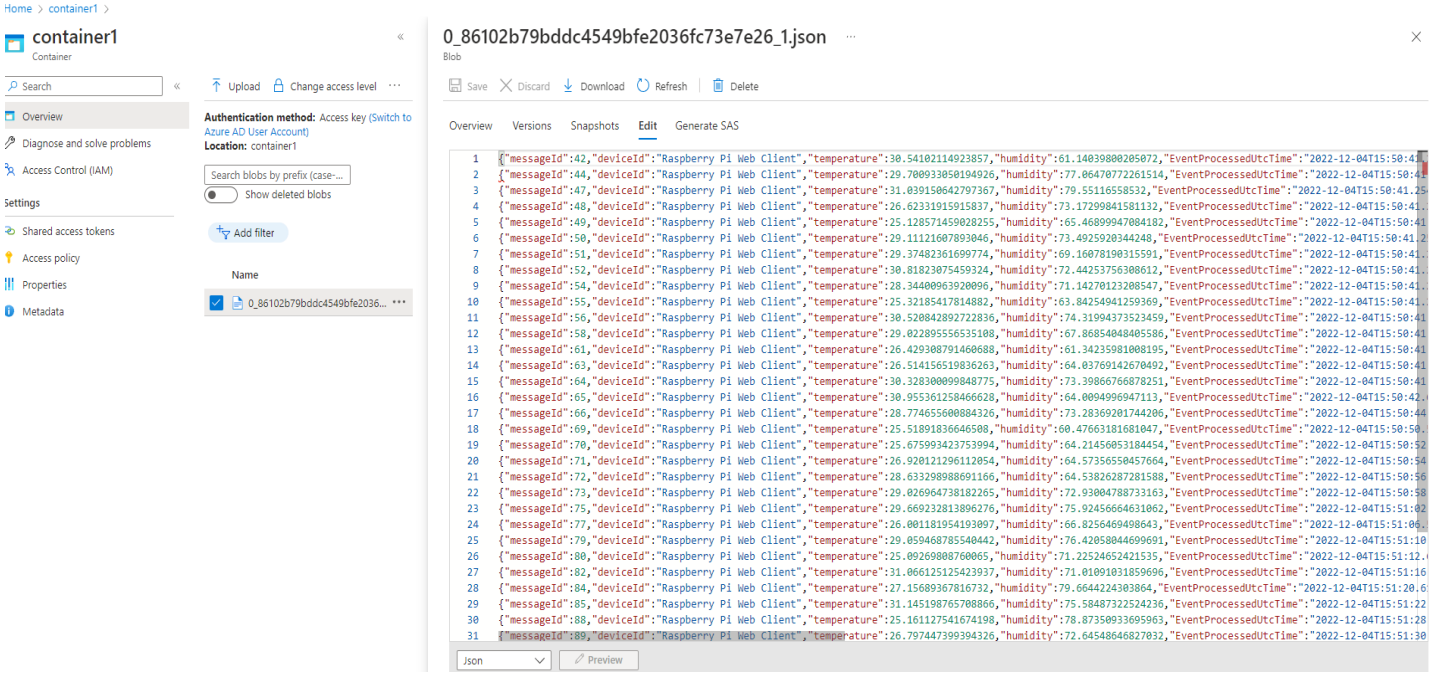
2. Explain how Stream Analytics works in Azure.

Azure Stream Analytics is a fully managed stream processing engine that is designed to analyze and process large volumes of streaming data with sub-millisecond latencies. Azure Stream Analytics connect to multiple sources and sinks, creating an end-to-end pipeline. Stream Analytics can connect to Azure Event Hubs and Azure IoT Hub for streaming data ingestion, as well as Azure Blob storage to ingest historical data. Job input can also include static or slow-changing reference data from Azure Blob storage or SQL Database that you can join to streaming data to perform lookup operations.

Stream Analytics can route job output to many storage systems such as Azure Blob storage, Azure SQL Database, Azure Data Lake Store, and Azure Cosmos DB. You can also run batch analytics on stream outputs with Azure Synapse Analytics or HDInsight, or you can send the output to another service, like Event Hubs for consumption or Power BI for real-time visualization.

3. Deploy all the resources in Azure Portal. Implement a Stream Analytics job by using the Azure portal.

The content in the container is shown below:



Part B

1. Explain what problem you are going to solve using this dataset. Provide a brief overview of your problem statement.

   Occupancy detection can be used as an input for HVAC control algorithms to save energy. A system built on machine learning algorithms that could accurately detect the presence of the occupants without using a camera can be truly valuable due to privacy concerns. The purpose of the project is to uses the temperature, Humidity, Light, CO2 and Humidity Ratio data to predict whether a property is occupied or not with azure machine learning. Data has 6 features date, temperature, Humidity, Light, CO2 and Humidity Ratio. The size of data is 20560 * 7, dataset has no NA values and needs minor preprocessing. The models i will try are Random Forest,and Logistic Regression.

2. Explain your dataset. Explore your dataset and provide at least 5 meaningful charts/graphs with an explanation.

Data has 6 features date, temperature, Humidity, Light, CO2 and Humidity Ratio. The size of data is 20560 * 7, dataset has no NA values. The object is on the Occupancy column, 1 means occupied, 0 means not occupied.

	date	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
0	2015-02-04 17:51:00	23.180000	27.272000	426.0	721.25	0.004793	1
1	2015-02-04 17:51:59	23.150000	27.267500	429.5	714.00	0.004783	1
2	2015-02-04 17:53:00	23.150000	27.245000	426.0	713.50	0.004779	1
3	2015-02-04 17:54:00	23.150000	27.200000	426.0	708.25	0.004772	1
4	2015-02-04 17:55:00	23.100000	27.200000	426.0	704.50	0.004757	1
...	...	...	...	...	...	...	...
20555	2015-02-04 10:38:59	24.290000	25.700000	808.0	1150.25	0.004829	1
20556	2015-02-04 10:40:00	24.330000	25.736000	809.8	1129.20	0.004848	1
20557	2015-02-04 10:40:59	24.330000	25.700000	817.0	1125.80	0.004841	1
20558	2015-02-04 10:41:59	24.356667	25.700000	813.0	1123.00	0.004849	1
20559	2015-02-04 10:43:00	24.408333	25.681667	798.0	1124.00	0.004860	1

20560 rows × 7 columns

```
1   df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20560 entries, 0 to 20559
Data columns (total 7 columns):
 #   Column         Non-Null Count  Dtype
---  ------         --------------  -----
 0   date           20560 non-null  object
 1   Temperature    20560 non-null  float64
 2   Humidity       20560 non-null  float64
 3   Light          20560 non-null  float64
 4   CO2            20560 non-null  float64
 5   HumidityRatio  20560 non-null  float64
 6   Occupancy      20560 non-null  int64
dtypes: float64(5), int64(1), object(1)
memory usage: 1.1+ MB
```

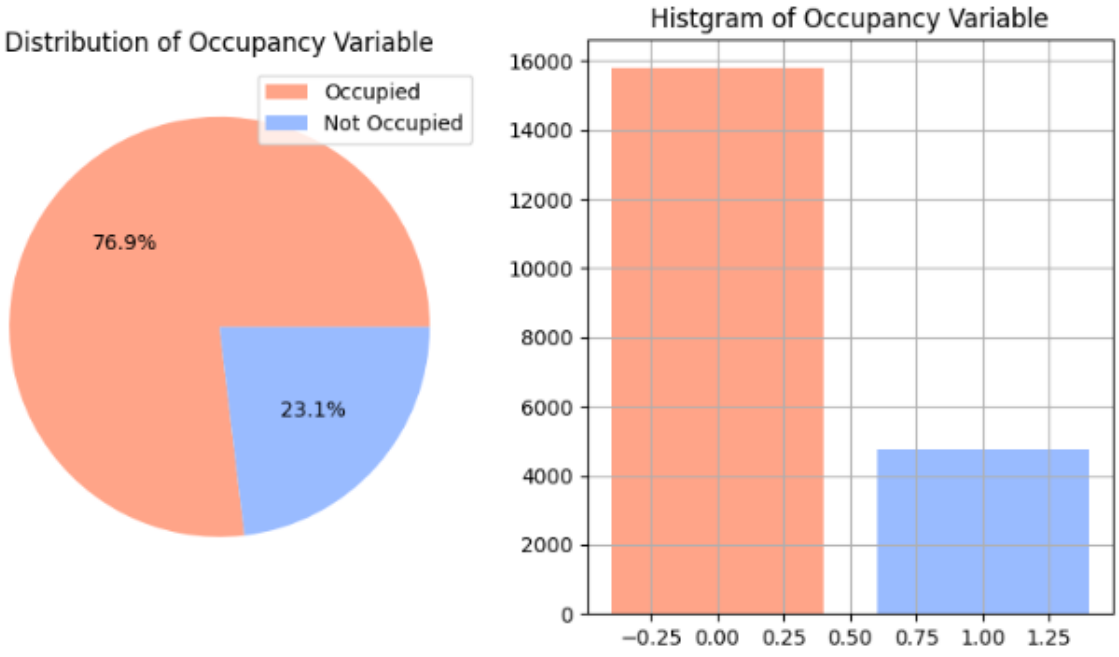
```
1   df.describe()
```

	Temperature	Humidity	Light	CO2	HumidityRatio	Occupancy
count	20560.000000	20560.000000	20560.000000	20560.000000	20560.000000	20560.000000
mean	20.906212	27.655925	130.756622	690.553276	0.004228	0.231031
std	1.055315	4.982154	210.430875	311.201281	0.000768	0.421503
min	19.000000	16.745000	0.000000	412.750000	0.002674	0.000000
25%	20.200000	24.500000	0.000000	460.000000	0.003719	0.000000
50%	20.700000	27.290000	0.000000	565.416667	0.004292	0.000000
75%	21.525000	31.290000	301.000000	804.666667	0.004832	0.000000
max	24.408333	39.500000	1697.250000	2076.500000	0.006476	1.000000

The bar chart shows 76.9% of the houses are occupied in the dataset, 23.1 of them are not occupied. The hist gram shows the number of occupied houses are close to 16000, unoccupied houses are around 4500 in the dataset.

```
bar_x = [0,1]
bar_height = df['Occupancy'].value_counts().values

plt.figure(figsize=(10,5))

# pie chart
plt.subplot(1, 2, 1)
plt.pie(df['Occupancy'].value_counts(),colors=['#FFA488','#99BBFF'],autopct='%1.1f%%')
plt.legend(['Occupied','Not Occupied'],loc='best')
plt.title('Distribution of Occupancy Variable')

# bar chart
plt.subplot(1, 2, 2)
plt.bar(x= bar_x,height= bar_height,
        color= ['#FFA488','#99BBFF'])
plt.title('Histgram of Occupancy Variable')
plt.grid()

plt.show()
```

✓ <1 sec



The seaborn plot demonstrate the distribution of each features by occupancy, we can tell that the temperature, light, CO2 features show a well-splitted distribution to occupied and unoccupied houses. While the humidity and humidityratio features have a chaotic distribution. Therefore, the temperature, light, CO2 features should have a better performance on predicting if a house is occupied or not.

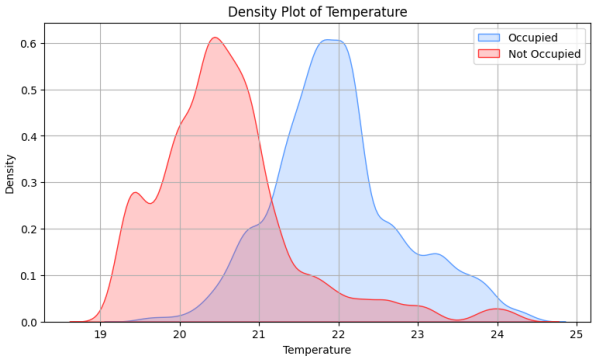
```
1    sns.pairplot(df, hue="Occupancy", height=3, diag_kind="kde")
```

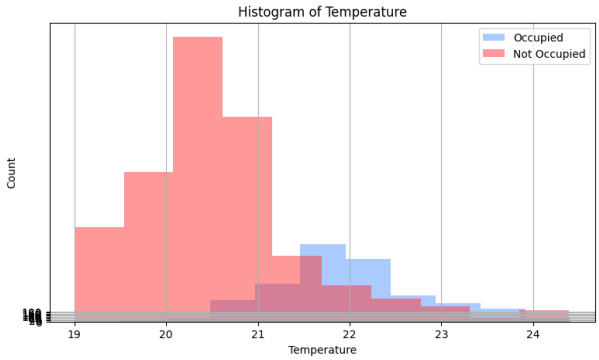
✓ 29 sec

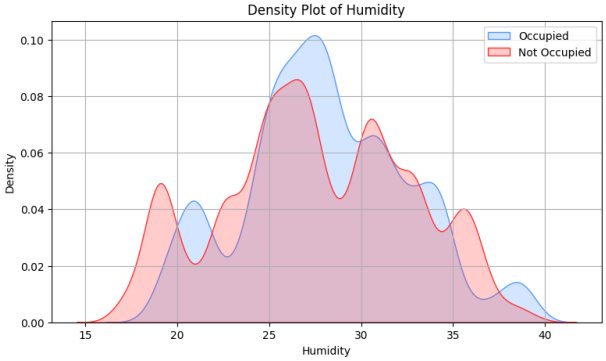
The code for density plot and hist gram plot of temperature is shown below. The code for drawing other features' plots is similar so only the plots are put in the report. We can conclude that the results are similar to the seaborn plot that the temperature, light, CO2 features should have a better performance on predicting if a house is occupied or not.

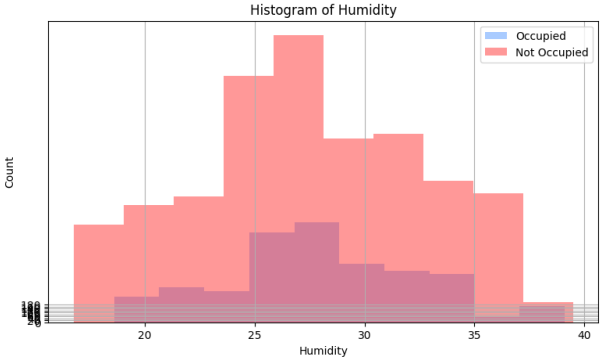
```
plt.figure(figsize=(20,5))

# density plot
plt.subplot(1, 2, 1)
# KDE plot of Occupancy
sns.kdeplot(df.loc[df['Occupancy'] != 0, 'Temperature'], label = 'target == 0',color='#5599FF',fill=True)

# KDE plot of healthy
sns.kdeplot(df.loc[df['Occupancy'] == 0, 'Temperature'], label = 'target == 1',color='#FF3333',fill=True)

# Labeling of plot
plt.xlabel('Temperature')
plt.ylabel('Density')
plt.legend(['Occupied','Not Occupied'],loc='upper right')
plt.title('Density Plot of Temperature')
plt.grid()

# histgram plot
plt.subplot(1, 2, 2)
plt.hist(df.loc[df['Occupancy'] != 0, 'Temperature'].reset_index(drop=True), alpha=0.5, label="Occupied",color='#5599FF')
plt.hist(df.loc[df['Occupancy'] == 0, 'Temperature'].reset_index(drop=True), alpha=0.5, label="Not Occupied",color='#FF3333')
plt.legend()
plt.xlabel('Temperature')
plt.ylabel('Count')
plt.yticks(np.arange(0, 200, 20))
plt.title('Histogram of Temperature')
plt.grid()


plt.show()
```

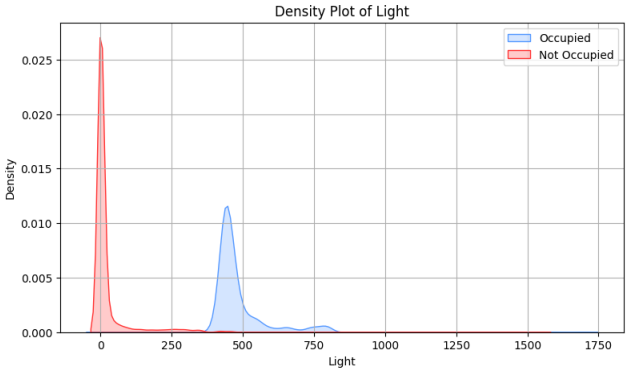
✓ <1 sec

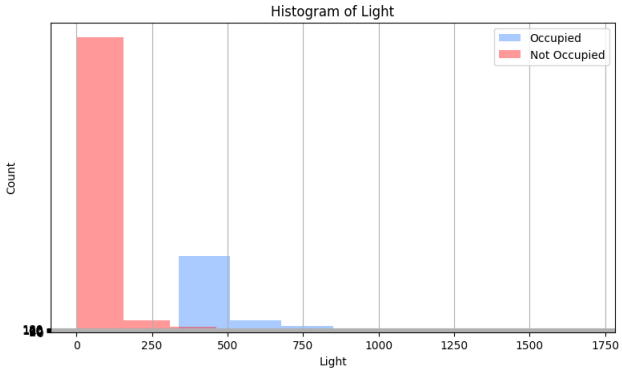


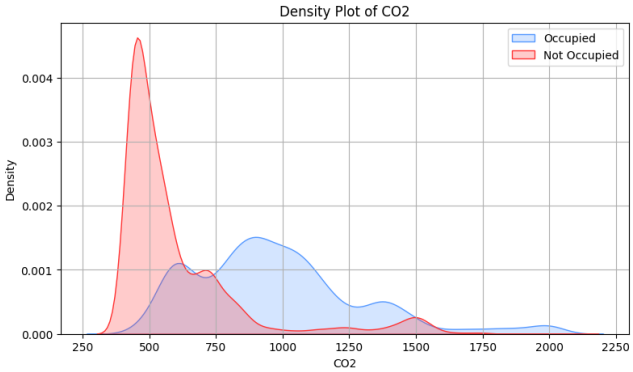


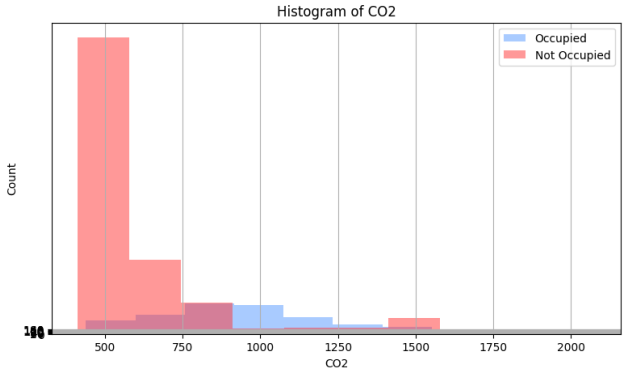


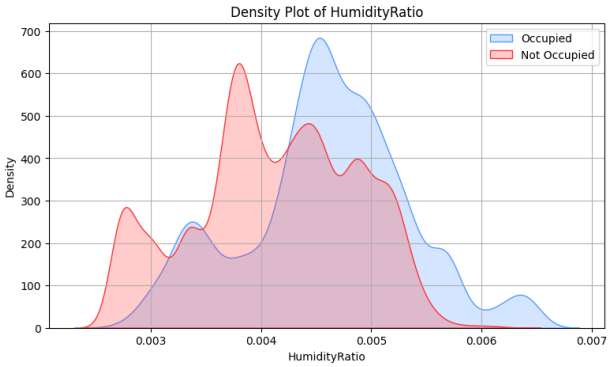


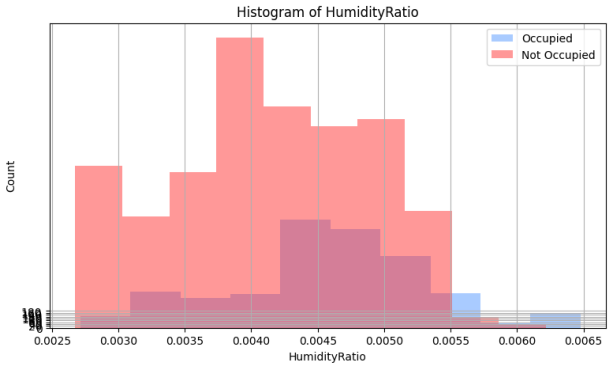












3. Do data cleaning/pre-processing as required and explain what you have done for your dataset and why?

   The dataset has a column date that is by instinct not relevant to occupancy, so we can drop the date column. Then to identify how important each feature is to predict the occupancy of a house; we use random forest classifier to compute the feature importance level of each feature. We found that Feature 1(Humidity) and Feature 4 (HumidityRatio) are the least important features that has minor effect on Occupancy, consider the total number of features is only 5, we choose to keep them in the model

Preprocessing

+ Code + Markdown

```
# Drop the date column that's not relevant to occupancy by instinct
df = df.drop(columns=['date'])
```

✓ <1 sec

```
# Select Features by Random Forest
X = df.drop(['Occupancy'],axis=1)
y = df['Occupancy']
feature_names = [f"feature {i}" for i in range(X.shape[1])]
forest = RandomForestClassifier(random_state=0)
forest.fit(X, y)

importances = forest.feature_importances_
std = np.std([tree.feature_importances_ for tree in forest.estimators_], axis=0)
forest_importances = pd.Series(importances, index=feature_names)
```

✓ 1 sec

+ Code + Markdown

```
forest_importances= forest_importances.sort_values(ascending=False)
forest_importances
```

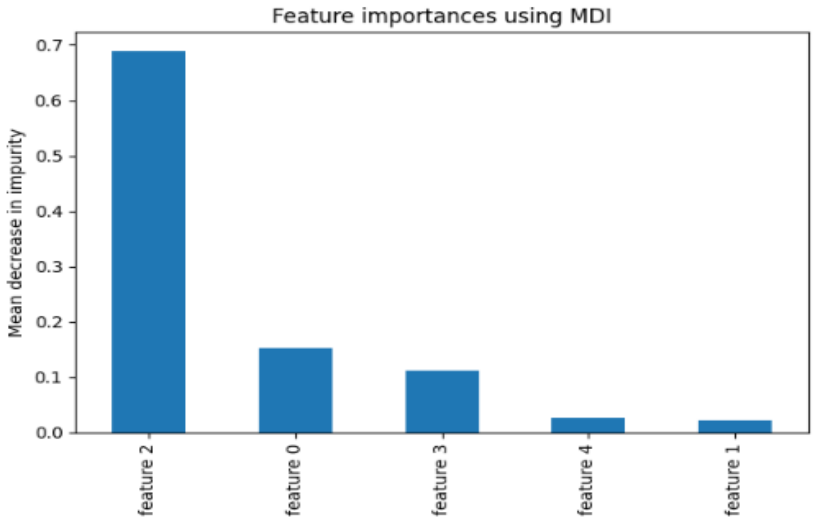
✓ <1 sec

```
feature 2    0.688343
feature 0    0.152295
feature 3    0.112118
feature 4    0.026418
feature 1    0.020826
dtype: float64
```

```
fig, ax = plt.subplots()
forest_importances.plot.bar() #yerr=std, ax=ax
ax.set_title("Feature importances using MDI")
ax.set_ylabel("Mean decrease in impurity")
fig.tight_layout()
```

:1]

✓ <1 sec



4. Implement 2 machine learning models and explain which algorithms you have selected and why. Compare them and show success metrics (Accuracy/RMSE/Confusion Matrix) as per your problem. Explain results.
5. Do hyperparameter tuning for your algorithms. Explain your results.

We first split the dataset into 70% train and 30% test data. Then use logistic regression to fit the train data. Logistic regression is an example of supervised learning. It is used to calculate or predict the probability of a binary (yes/no) event occurring. In this case it is used for classfication of weather a house is occupied or not. The accuracy of training without any tuning is 98.88% which is really good, applying it on the test set, the accuracy is 98.91%. The test result is a little more accurate but can be neglected so the model is not overfitting or underfitting.

```
# split the train test
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

✓ <1 sec

```
lr = LogisticRegression()
lr.fit(X_train,y_train)
print("accuracy score in train dataset :",lr.score(X_train, y_train))
print("accuracy score in test dataset :",lr.score(X_test, y_test))
```

✓ <1 sec

```
accuracy score in train dataset : 0.9888827126181212
accuracy score in test dataset : 0.9891374837872893
```

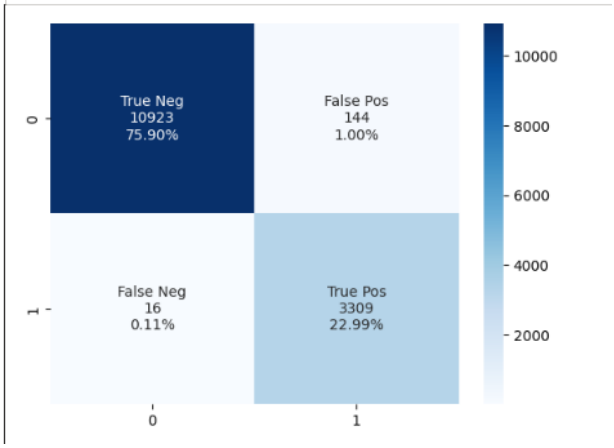
The confusion matrix function is defined.

```
#confusion matrix
def create_cf(matrix):
  cf_matrix = matrix
  group_names = ['True Neg','False Pos','False Neg','True Pos']
  group_counts = ["{0:0.0f}".format(value) for value in
                  cf_matrix.flatten()]
  group_percentages = ["{0:.2%}".format(value) for value in
                       cf_matrix.flatten()/np.sum(cf_matrix)]
  labels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in
            zip(group_names,group_counts,group_percentages)]
  labels = np.asarray(labels).reshape(2,2)
  sns.heatmap(cf_matrix, annot=labels, fmt='', cmap='Blues')
```

From the confusion matrix plot, the false positive rate is 1%, and the false negative rate is only 0.11%. The logistic regression model without any tuning is already very accurate.

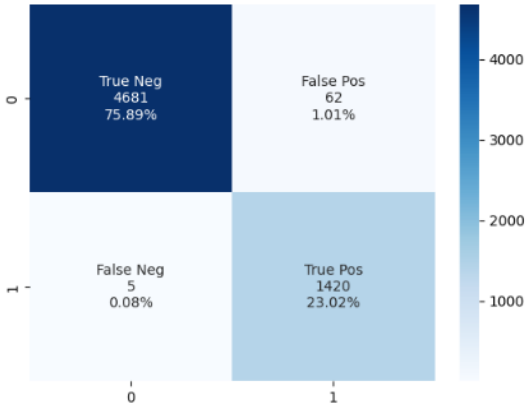
```
# Trainging CF Matrix
y_pre1 = lr.predict(X_train)
y_pre2 = lr.predict(X_test)
create_cf(confusion_matrix(y_train, y_pre1))
```

✓ <1 sec



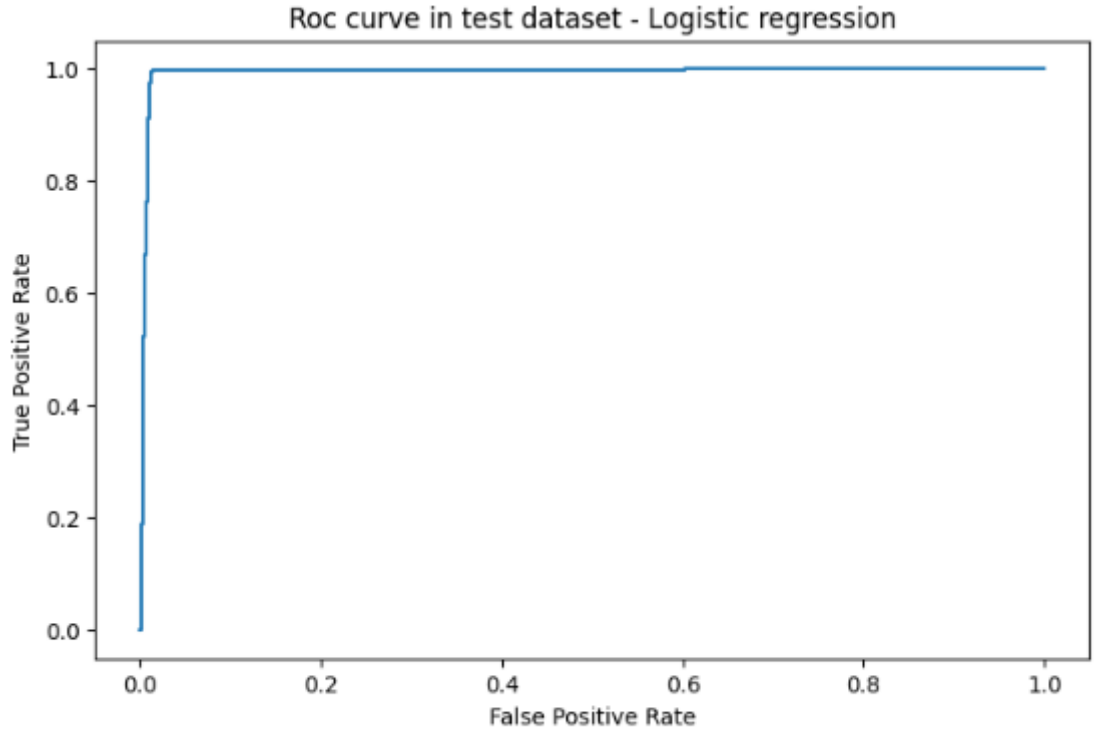

```
# Testing CF Matrix
create_cf(confusion_matrix(y_test, y_pre2))
```

✓ <1 sec




```
#ROC curve
y_pro2 = lr.predict_proba(X_test)[:,1]
false_positive_rate, true_positive_rate, threshold = roc_curve(y_test, y_pro2)
plt.figure(figsize=(8,5))
plt.plot(false_positive_rate, true_positive_rate)
plt.title('Roc curve in test dataset - Logistic regression')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
print("roc_auc_score :", roc_auc_score(y_test, y_pro2))
```

✓ <1 sec




```
roc_auc_score : 0.9946712237054792
```

The logistic regression model has different solvers, and hyperparameter C can be tuned. Grid search is used to find the optimal solver and C value. As a result, best C is 0.2616 and the best solver is liblinear. The training accuracy increased 0.01% after tuning, and the testing accuracy dropped 0.02%.

Logistic Regression Model Tuning

```
[29]
#grid search for hyper-parameter C and solver
grid = dict()
grid['solver'] = ['saga', 'liblinear']
grid['C'] = np.linspace(0, 5, 20)
log = LogisticRegression()
logreg_cv = GridSearchCV(log, grid, cv=10)
logreg_cv.fit(X_train,y_train)

print("best parameters: ",logreg_cv.best_params_)
print("accuracy :",logreg_cv.best_score_)
```

✓ 32 sec

```
best parameters:  {'C': 0.2631578947368421, 'solver': 'liblinear'}
accuracy : 0.9889520596865106
```

+ Code + Markdown

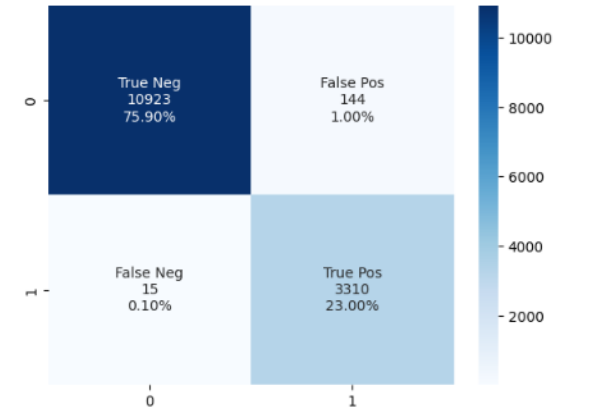
```
[30]
# impletment model with best C and solver
best_c  = logreg_cv.best_params_.get("C")
best_solver = logreg_cv.best_params_.get("solver")
log_best = LogisticRegression(penalty='l1', solver = best_solver, C=best_c)
log_best.fit(X_train,y_train)
print("accuracy score in train dataset :",log_best.score(X_train, y_train))
print("accuracy score in test dataset :",log_best.score(X_test, y_test))
```

✓ <1 sec

```
accuracy score in train dataset : 0.988952195664258
accuracy score in test dataset : 0.9891374837872893
```

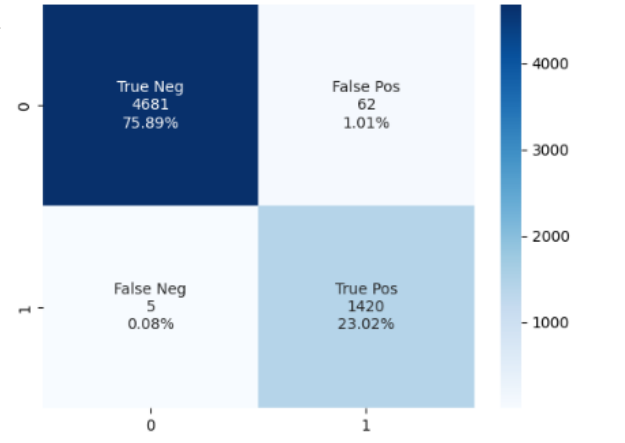
```
# Trainging CF Matrix
y_pre3 = log_best.predict(X_train)
y_pre4 = log_best.predict(X_test)
create_cf(confusion_matrix(y_train, y_pre3))
```

✓ <1 sec




```
# Testing CF Matrix
create_cf(confusion_matrix(y_test, y_pre4))
```

✓ <1 sec



The random forest classifier model is used to predict the occupancy of houses. Random forest is a Supervised Machine Learning Algorithm that is used widely in Classification and Regression problems. The accuracy of training without any tuning is 100% which is perfect, applying it on the test set, the accuracy is 99.27%. The test result is also close to 100% which means the model is not overfitting or underfitting.

Random Forest Model

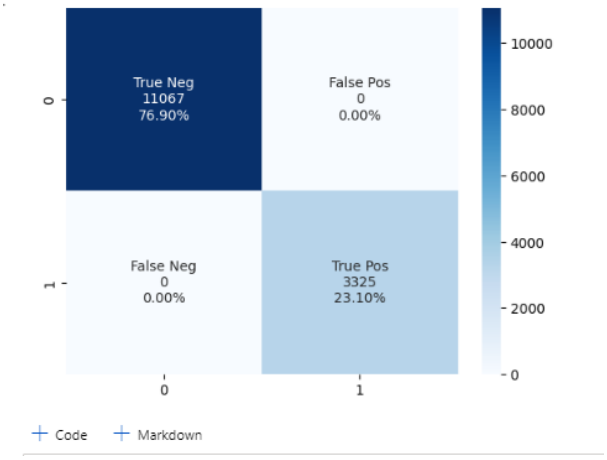
```
3]  1  from sklearn.ensemble import RandomForestClassifier
    ✓ <1 sec
```

```
4]  1  forest = RandomForestClassifier(random_state=0)
    2  forest.fit(X_train, y_train)
    3  print("accuracy score in train dataset :",forest.score(X_train, y_train))
    4  print("accuracy score in test dataset :",forest.score(X_test, y_test))
    ✓ <1 sec
```

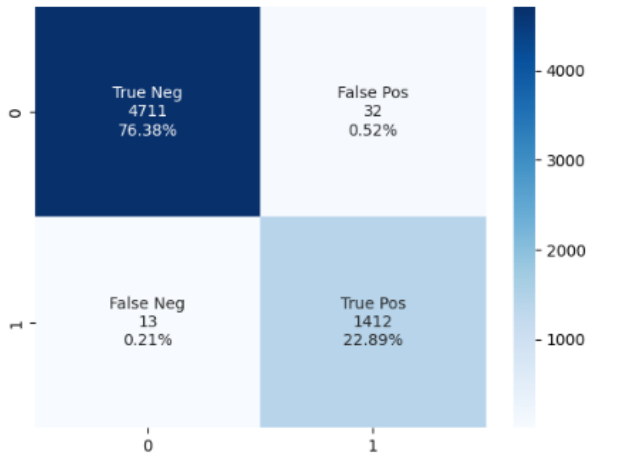
```
accuracy score in train dataset : 1.0
accuracy score in test dataset : 0.992704280155642
```

The confusion matrix on the test data shows the false positive rate is 0.52% and the false negative rate is 0.21%. The error is acceptable as is without any tuning on the model.

```
1  # Trainging CF Matrix
2  y_pre5 = forest.predict(X_train)
3  y_pre6 = forest.predict(X_test)
4  create_cf(confusion_matrix(y_train, y_pre5))
✓ <1 sec
```



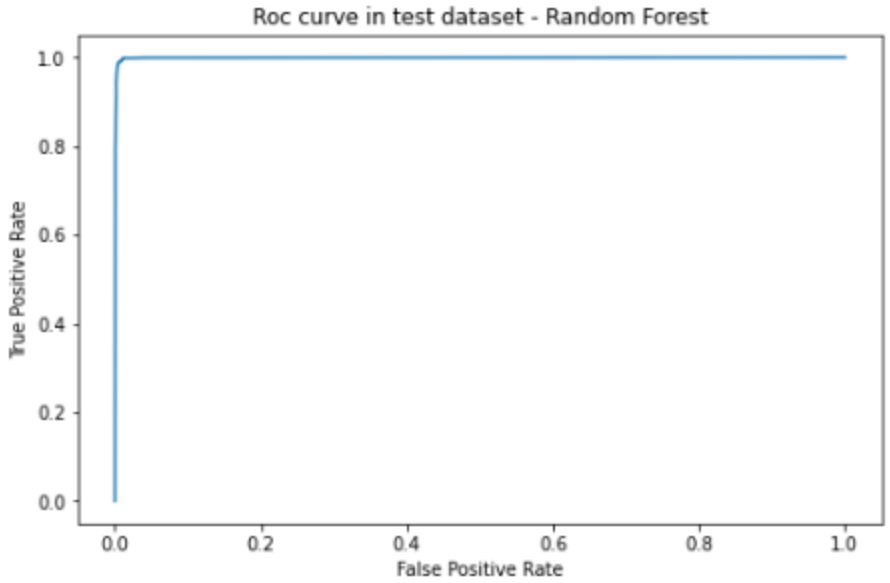

```
1  # Testing CF Matrix
2  create_cf(confusion_matrix(y_test, y_pre6))
✓ <1 sec
```



+ Code + Markdown

The ROC curve on the random forest model shows 99.90% AUC score

```
#ROC curve
y_pro3 = forest.predict_proba(X_test)[:,1]
false_positive_rate, true_positive_rate, threshold = roc_curve(y_test, y_pro3)
plt.figure(figsize=(8,5))
plt.plot(false_positive_rate, true_positive_rate)
plt.title('Roc curve in test dataset - Random Forest')
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
print("roc_auc_score :", roc_auc_score(y_test, y_pro3))
```




```
roc_auc_score : 0.9990134306882533
```

The n_estimators and the max features parameters can be tuned in random forest classifier model. using the grid search, sqrt is the best max features and the best n_estimators is 100. It is interesting that the best model after turning shows the same result as the original random forest model without any tuning. Where the accuracy for train data is 100% and the accuracy for test data is 99.27%. The result makes sense because the training accuracy is already 100%, there is no room for improvement of the original model.

```
[38]
#grid search for hyper-parameter C and solver
grid = dict()
grid['n_estimators'] = [2,5,10,20,50,100,200]
grid['max_features'] = ['sqrt', 'log2', None]
rf = RandomForestClassifier(random_state=0)
rf_cv = GridSearchCV(rf, grid, cv=10)
rf_cv.fit(X_train,y_train)

print("best parameters: ",rf_cv.best_params_)
print("accuracy :",rf_cv.best_score_)
✓ 2 min 26 sec

best parameters:  {'max_features': 'sqrt', 'n_estimators': 100}
accuracy : 0.992078700486449
```

```
# impletment model with best C and solver
best_n_estimators  = rf_cv.best_params_.get("n_estimators")
best_max_features = rf_cv.best_params_.get("max_features")
rf_best = RandomForestClassifier(random_state=0, max_features = best_max_features, n_estimators=best_n_estimators)
rf_best.fit(X_train,y_train)
print("accuracy score in train dataset :",rf_best.score(X_train, y_train))
print("accuracy score in test dataset :",rf_best.score(X_test, y_test))
```

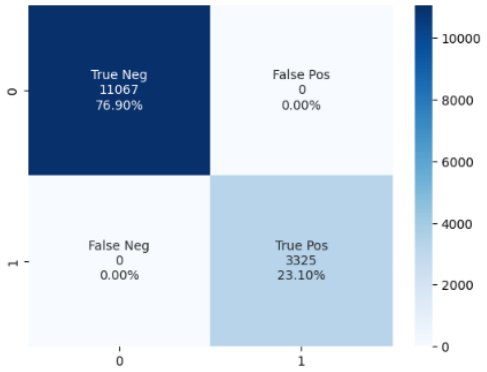
✓ <1 sec

```
accuracy score in train dataset : 1.0
accuracy score in test dataset : 0.992704280155642
```

[40]

```
# Trainging CF Matrix
y_pre7 = forest.predict(X_train)
y_pre8 = forest.predict(X_test)
create_cf(confusion_matrix(y_train, y_pre7))
```

✓ <1 sec




```
# Testing CF Matrix
create_cf(confusion_matrix(y_test, y_pre8))
```

✓ <1 sec