Aim: To perform Classification modelling on given dataset

Theory:

Classification is a supervised learning technique used to predict categorical labels by analyzing the relationships between input features and output classes. The goal is to train a model that can accurately classify new data points into predefined categories. Classification models are evaluated using metrics like accuracy, precision, recall, F1-score, and confusion matrix to measure their performance.

Types of Classification Algorithms:

1. **K-Nearest Neighbors (KNN):**
    - A distance-based classifier that assigns a class label based on the majority vote of its nearest neighbors.
    - Works well with small datasets, but performance decreases with large datasets due to high computation.
    - Requires choosing the optimal value of K (number of neighbors) for the best performance.
    - Sensitive to feature scaling, so normalization is necessary.

2. **Naive Bayes:**
    - A probabilistic classifier based on Bayes' Theorem, assuming independence between features.
    - Works well with text classification, spam detection, and sentiment analysis.
    - Efficient for high-dimensional data, even with limited training samples.
    - Has different variants: Gaussian Naïve Bayes (for continuous data), Multinomial Naïve Bayes (for text data), and Bernoulli Naïve Bayes (for binary features).

3. **Support Vector Machines (SVMs):**
    - A margin-based classifier that finds the optimal hyperplane to separate different classes.
    - Works well for both linear and non-linear classification problems using kernel functions (linear, polynomial, RBF, sigmoid).
    - Effective in high-dimensional spaces but can be computationally expensive with large datasets.
    - Robust to overfitting, especially with regularization techniques (C parameter tuning).

4. **Decision Tree:**

- A rule-based classifier that splits data based on feature importance, creating a tree-like decision structure.
- Easy to interpret and visualize but prone to overfitting if not pruned.
- Handles both numerical and categorical data well.
- Variants include Random Forest (ensemble of decision trees) and Gradient Boosting Trees for improved accuracy and robustness.

Steps:

1. **Load the dataset**

```
# Load the dataset
file_path = "/content/train.csv"
df = pd.read_csv(file_path)

# Display first few rows
df.head()
```

	Unnamed: 0	id	Gender	Customer Type	Age	Type of Travel	Class	Flight Distance	Inflight wifi service	Departure/Arrival time convenient	...	Inflight entertainment	On-board service	Leg room service	Baggage handling	Checkin service	Inflight service	Cleanliness	Depa Del Mi
0	0	70172	Male	Loyal Customer	13	Personal Travel	Eco Plus	460	3	4	...	5	4	3	4	4	5	5	
1	1	5047	Male	disloyal Customer	25	Business travel	Business	235	3	2	...	1	1	5	3	1	4	1	
2	2	110028	Female	Loyal Customer	26	Business travel	Business	1142	2	2	...	5	4	3	4	4	4	5	
3	3	24026	Female	Loyal Customer	25	Business travel	Business	562	2	5	...	2	2	5	3	1	4	2	
4	4	119299	Male	Loyal Customer	61	Business travel	Business	214	3	3	...	3	3	4	4	3	3	3	

The dataset consists of 103,904 entries with 23 columns, capturing various aspects of airline passengers' experiences and satisfaction levels. It contains a mix of categorical and numerical variables. The categorical attributes include Gender, Customer Type, Type of Travel, Class, and Satisfaction, which provide insights into passenger demographics and travel preferences. The numerical attributes include Age, Flight Distance, Departure and Arrival Delays, and various in-flight service ratings such as WiFi service, food and drink, seat comfort, and baggage handling, all measured on an integer scale. The dataset appears to be complete, with no missing values, making it well-suited for exploratory data analysis and predictive modeling.

2. **Data preprocessing and splitting**

```
# Encode categorical variables
label_encoders = {}
categorical_cols = ["Gender", "Customer Type", "Type of Travel", "Class", "satisfaction"]

for col in categorical_cols:
    le = LabelEncoder()
    df[col] = le.fit_transform(df[col])  # Convert to numerical labels
    label_encoders[col] = le  # Store encoder for inverse transformation if needed later

# Separate features and target
X = df.drop("satisfaction", axis=1)
y = df["satisfaction"]

# Normalize numerical features for KNN & SVM
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# Split into training & test sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2, random_state=42, stratify=y)

# Check data shapes
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

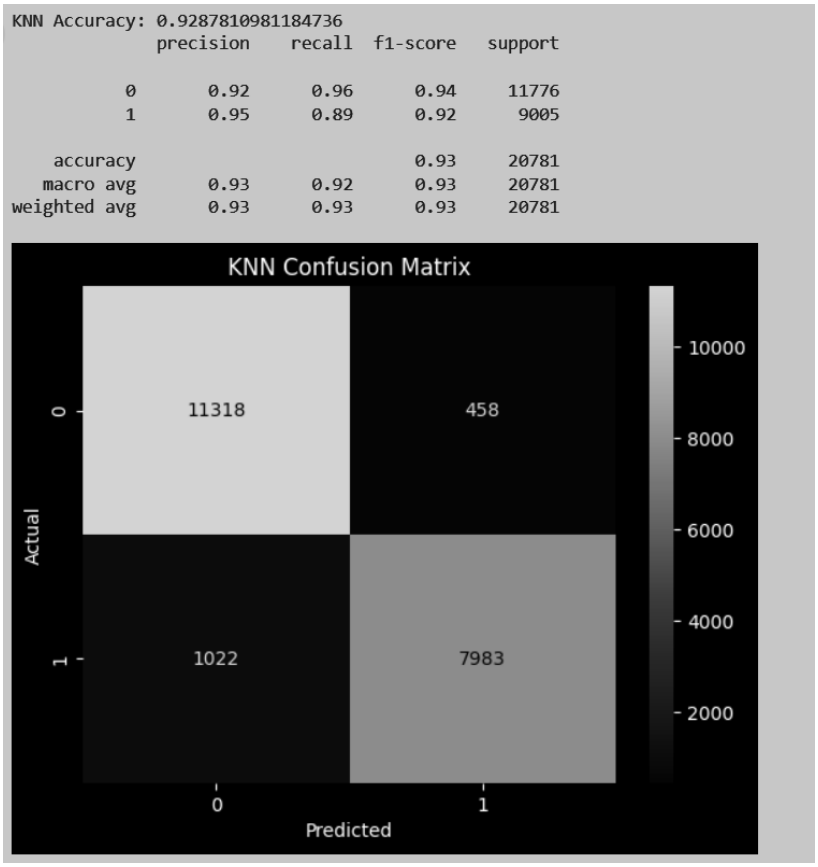
We preprocess the dataset for classification by encoding categorical variables using LabelEncoder, separating features and the target variable (satisfaction), and normalizing numerical features using StandardScaler—important for models like KNN and SVM. **The dataset is then split into training (80%) and testing (20%**) sets using train_test_split, ensuring balanced class distribution with stratification

3. **KNN model**

```
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)

y_pred_knn = knn.predict(X_test)

print("KNN Accuracy:", accuracy_score(y_test, y_pred_knn))
print(classification_report(y_test, y_pred_knn))

sns.heatmap(confusion_matrix(y_test, y_pred_knn), annot=True, fmt="d", cmap="Blues")
plt.title("KNN Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

We apply the K-Nearest Neighbors (KNN) classifier with n_neighbors=5 to classify data. The model is trained on X_train and y_train using knn.fit(), then tested on X_test to generate predictions (y_pred_knn).

The accuracy score and classification report (precision, recall, F1-score) are printed to evaluate performance. Additionally, a confusion matrix is computed and visualized using sns.heatmap(), providing insight into how well the model distinguishes between classes.



Accuracy: The model achieved 92.88%, showing strong performance.

Precision & Recall:
Class 0: High recall (0.96) indicates fewer false negatives.
Class 1: Slightly lower recall (0.89) suggests some misclassification.

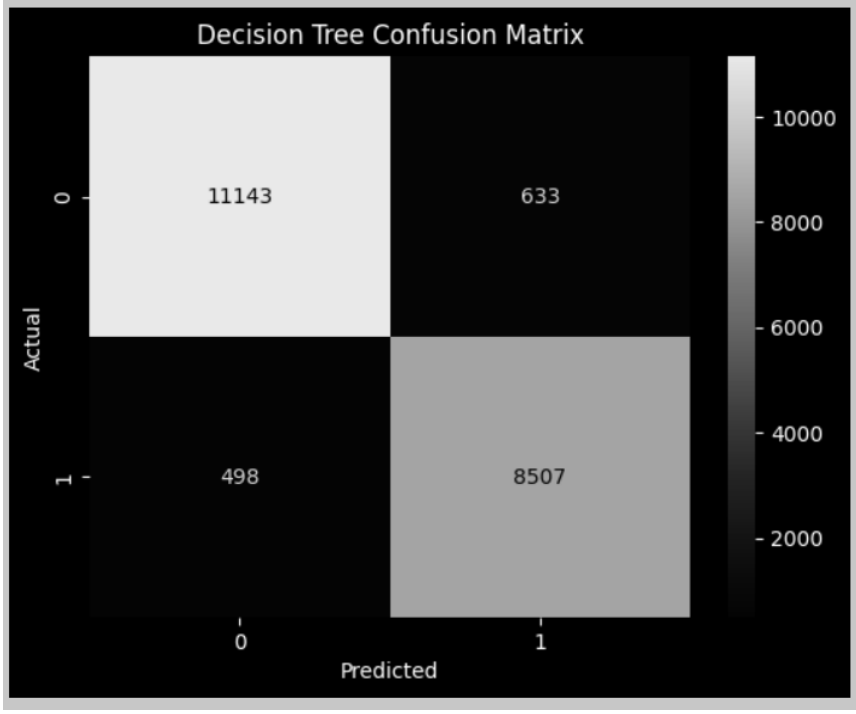
TP: 7983, TN: 11318.
FP: 458, FN: 1022.

4. **Decision Tree**

```
dt = DecisionTreeClassifier(random_state=42)
dt.fit(X_train, y_train)

y_pred_dt = dt.predict(X_test)


print("Decision Tree Accuracy:", accuracy_score(y_test, y_pred_dt))
print(classification_report(y_test, y_pred_dt))

sns.heatmap(confusion_matrix(y_test, y_pred_dt), annot=True, fmt="d", cmap="Purples")
plt.title("Decision Tree Confusion Matrix")
plt.xlabel("Predicted")
plt.ylabel("Actual")
plt.show()
```

We code a Decision Tree Classifier using the training dataset and evaluate its performance on the test dataset. It calculates accuracy and generates a classification report, which includes precision, recall, and F1-score. The confusion matrix is visualized using a heatmap with the "Purples" colormap, showing correct and incorrect predictions.

```
Decision Tree Accuracy: 0.9455752851162119
              precision    recall  f1-score   support

           0       0.96      0.95      0.95     11776
           1       0.93      0.94      0.94      9005

    accuracy                           0.95     20781
   macro avg       0.94      0.95      0.94     20781
weighted avg       0.95      0.95      0.95     20781
```



1. The Decision Tree Classifier achieved an accuracy of 94.56%, performing well in classifying both classes. The confusion matrix shows 11143 true negatives and 8507 true positives, indicating strong predictive capability. **However, 633 false positives and 498 false negatives suggest some misclassification.**
2. The precision and recall values are balanced, indicating a good trade-off between false positives and false negatives. Compared to KNN, the Decision Tree performs slightly better, likely due to its ability to capture complex decision boundaries.
3. We also tried to apply the **SVM model** on the same dataset. But the process was unsuccessful, it always took a very long time to train the model and no favourable outcome could be obtained.
4. Since **SVM solves a quadratic programming problem**, which is computationally expensive compared to other models like Decision Trees or KNN, it is possible that due to the size of our dataset, the computational overhead increased very much. Since the dataset has a large number of instances and features, SVM takes a long time to give favorable outcomes.

Conclusion:

From the previous experiments we understood how regression helps to understand the data, But here the right classification model depends on dataset size, computational efficiency, and feature distribution. **KNN performed well with 92.88% accuracy** but struggled slightly with class imbalance. **Decision Tree outperformed KNN with 94.56% accuracy**, effectively capturing decision boundaries while maintaining a balance between precision and recall. However, **SVM proved impractical due to high running time**, making it unsuitable for large datasets. The experiment highlights the importance of feature selection, preprocessing, and model interpretability in achieving optimal classification results.