Dugar-Arihant-Tirumala_Vinjamuri-Abhinav_Acharya-Assignment-1

September 21, 2023

[1]:
```
import sklearn
from sklearn.datasets import load_breast_cancer
from sklearn.model_selection import train_test_split
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score
from sklearn.model_selection import KFold
from sklearn.manifold import TSNE
import numpy as np
import matplotlib.pyplot as plt
import plotly.express as px
import pandas as pd
```

[2]:
```
#Load breast cancer dataset
X, y = load_breast_cancer(return_X_y=True)
```

REFERENCES:

[1] "Sklearn.Datasets.Load_linnerud," scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_linnerud.html. [Accessed: 21-Sep-2023].

EXPLANATION:

return_X_y= True returns data and target as a tuple of separate arrays instead of a combined object from which we need to assign X and y.

when return_X_y=True, the output is `(data, target)` when return_X_y=False, the output is a combined object, for example: `Data = load_breast_cancer()` and we need to access data and target like `X = Data.data`, and `y = Data.target`

[3]:
```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)
```

REFERENCES:

[2] "Glossary of common terms and API elements," scikit-learn. [Online]. Available: https://scikit-learn.org/stable/glossary.html. [Accessed: 21-Sep-2023].

EXPLANATION:

Test size is defined so that training happens on 80% of the data, and the test on 20% of the data. A fixed value of random_state is used to reproduce consistent results in different runs. We could leave it undefined to produce different splits each time.

[4]:
```
# Create a Random Forest Classifier
rf_classifier = RandomForestClassifier(max_depth=10, random_state=42)

# Train the classifier on the training data
rf_classifier.fit(X_train, y_train)

# Make predictions on the test set
y_pred = rf_classifier.predict(X_test)

# Calculate the accuracy of the classifier on the test set
accuracy = accuracy_score(y_test, y_pred)

print(f"Accuracy of the Random Forest Classifier: {accuracy:.2f}")
```

```
Accuracy of the Random Forest Classifier: 0.96
```

REFERENCES:

[3] "Sklearn.Ensemble.RandomForestClassifier," scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html. [Accessed: 21-Sep-2023].

[4] R. Ragini, "Principle of parsimony," Medium, 06-Jun-2019. [Online]. Available: https://medium.com/@ruhi3929/principle-of-parsimony-d510356ca06a. [Accessed: 21-Sep-2023].

EXPLANATION:

1. In the first line of code, we created a classifier with a bunch of defined parameters:
   - `max_depth` predefines the depth of the decision trees. This is done to avoid overfitting of the model by prepruning the tree depth (size).
   - A fixed value of `random_state` is used to reproduce consistent results in different runs.
2. In the second line of the code, we are fitting the classifier to our training data, to make sure that the model learns from the training data.
3. Now that the classifier is ready, we are predicting the lables for the test data in the third line.
4. At last, we are calculating the accuracy of the classifier by comparing the predicted labels (`y_pred`) to the actual labels (`y_test`)

We tried checking the accuracy of the model without setting the hyperparameter – max_depth. The accuracy turned out to be 0.96. Then we gradually experimented with the hyperparameter to see if we could find the simplest solution with lesser assumptions made (Law of Parsimony). We have noticed that using max_depth = 10, the accuracy was still the same. So we had proceeded with 10 as it was the simplest solution with fewer assumptions made.).

[5]:
```
# Creating classifier using loops
acc_scores = []
```

```
clf = RandomForestClassifier(n_estimators=100, max_depth=10,␣
 ↪criterion='entropy')

# 10 different splits
for i in range(10):
    X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=i)
    clf.fit(X_train, y_train)
    y_pred = clf.predict(X_test)
    acc = accuracy_score(y_test, y_pred)
    acc_scores.append(acc)

acc_mean = np.mean(acc_scores)
acc_std = np.std(acc_scores)

print(f'Mean Accuracy: {acc_mean:.2f}')
print(f'Standard Deviation: {acc_std:.2f}')
```

```
Mean Accuracy: 0.96
Standard Deviation: 0.02
```

EXPLANATION:

We do the same thing that we did till the previous steps, except for this case, we do it in a for loop with a range of 10, and append the calculated accuracy scores to an array at the end to get the avg mean and std. A high avg. mean value indicates that the model can predict accurately on an average. A low avg. std value indicates that the model does not vary much in its executions accross different splits of the data. More over, a varying random state enables us to avoid bias of the model as it produces varying mean values across different executions due to the fact that the model runs on different subsets of data each time. A fixed random state results in same mean values and very low standard deviation.

[6]:
```
#This is using KFold Generator
accuracies = []

# Create a KFold generator
kf = KFold(n_splits=10, random_state=42, shuffle=True)

# Iterate through different splits
for train_index, test_index in kf.split(X):
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    rf_classifier.fit(X_train, y_train)
    y_pred = rf_classifier.predict(X_test)
    # Calculate the accuracy of the classifier on the test set
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)
```

```
# Calculate the mean and standard deviation of accuracies
mean_accuracy = np.mean(accuracies)
std_accuracy = np.std(accuracies)

print(f"Mean Accuracy: {mean_accuracy:.2f}")
print(f"Standard Deviation of Accuracy: {std_accuracy:.2f}")
```

```
Mean Accuracy: 0.96
Standard Deviation of Accuracy: 0.01
```

REFERENCES:

[5] "Sklearn.Model_selection.KFold," scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.KFold.html. [Accessed: 21-Sep-2023].

EXPLANATION:

We are using the KFold to split into 10 parts for cross-validation. The accuracies are appended at the end into the array to calculate the mean and average. A high avg. mean value indicates that the model can predict accurately on an average. A low avg. std value indicates that the model does not vary much in its executions accross different splits of the data. Observing the above two solutions, we see that KFold has lower standard deviation which indicates that it is more efficient but it requires more computation power.

[7]:
```
scaler = sklearn.preprocessing.StandardScaler()
X_std = scaler.fit_transform(X)

# Create PCA instance
pca = sklearn.decomposition.PCA(n_components=2)

# Fit PCA to standardized features
pca.fit(X_std)

# Transform standardized features using PCA
X_pca = pca.transform(X_std)
```

REFERENCES:

[6] "Sklearn.decomposition.PCA," scikit-learn. [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html. [Accessed: 21-Sep-2023].

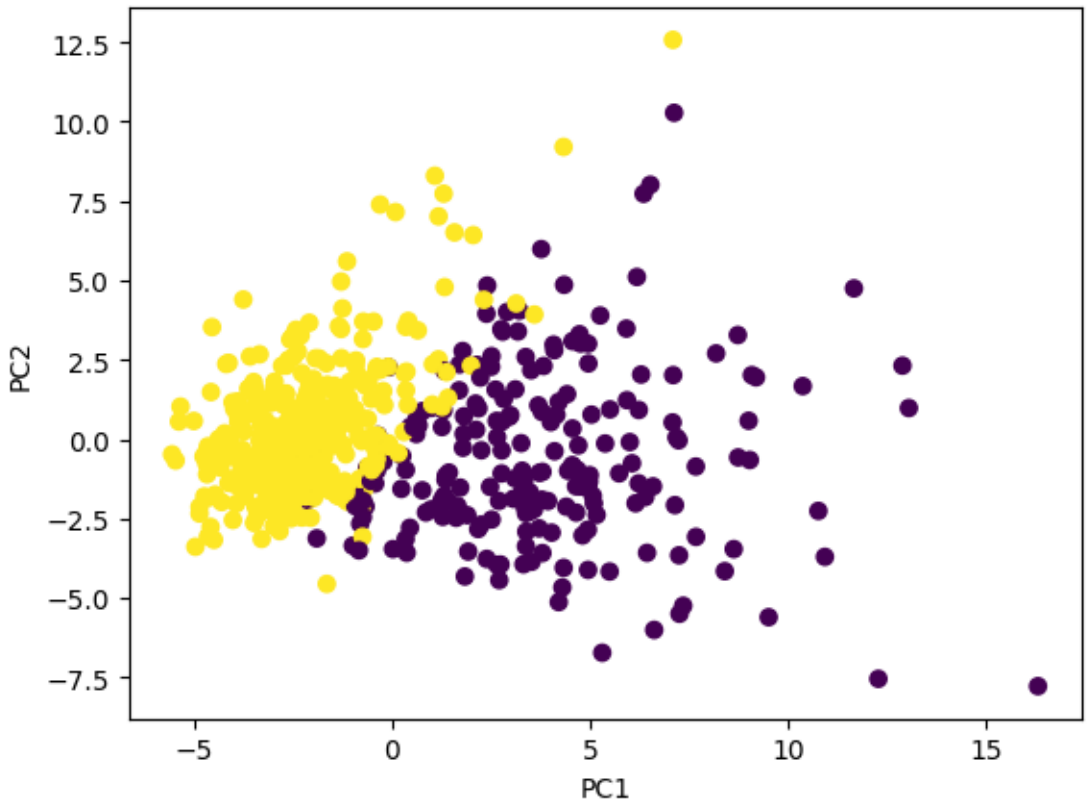
EXPLANATION:

Here we are trying to condense down the features of the original dataset (since there could be many columns). Visualizing large datasets with many dimensions can be harder, which is why we need to condense the dimensions to a 2d or a 3d plot. We need to make sure that we dont lose information while condensing the dimensions.

We are creating a standard scaler instance first, and then perform `fit_transform()` to learn the dimensions and then transform it by calculating the mean and std (fit) and then standardizing the data by subtracting the mean and dividing the std (transform). Then we are creating a PCA

instance by defining the number of dimensions as 2 to reduce the data down to 2 dimensions. Then using `fit()` we calculate the principle component axes where the data varies the most. Then atlast, we perform `transform()` to reduce the dimensionality down to 2. It projects the data point onto the first 2 principle component axis, resulting in a new dataset with 2 features (dimensions).

[8]:
```
plt.scatter(X_pca[:,0], X_pca[:,1], c=y)
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.show()
```

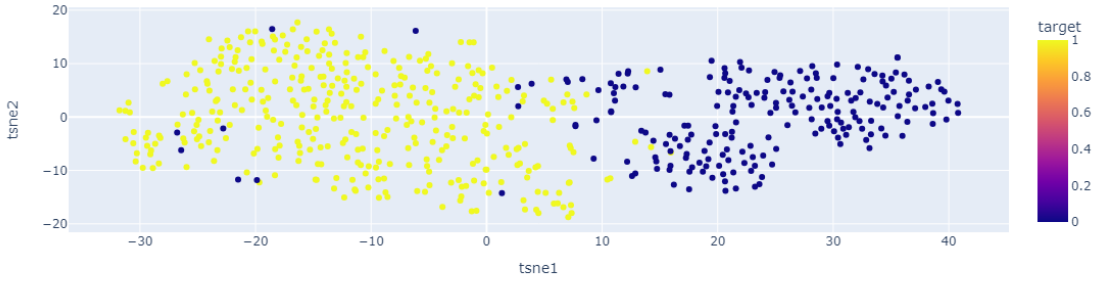


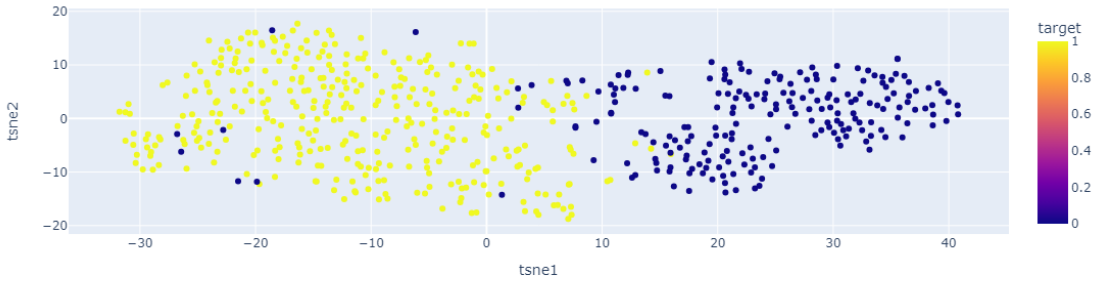
EXPLANATION:

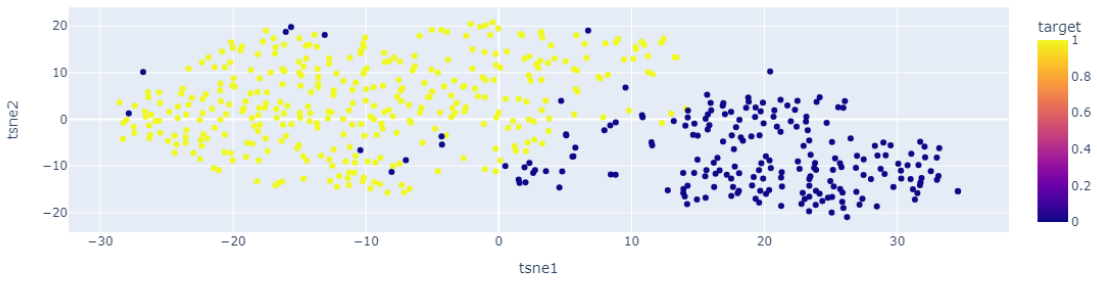
We can cleary see that both the classes are easily separable and distinguishable (with a little overlap) with the current principle components. Class A has very little variance accross both the principle component axes(little spread) compared to class B which has a higher variance accross principle component axis 1(horizontally spread). We can notice a few outliers in both classes which could be considered as unusualities in the data.

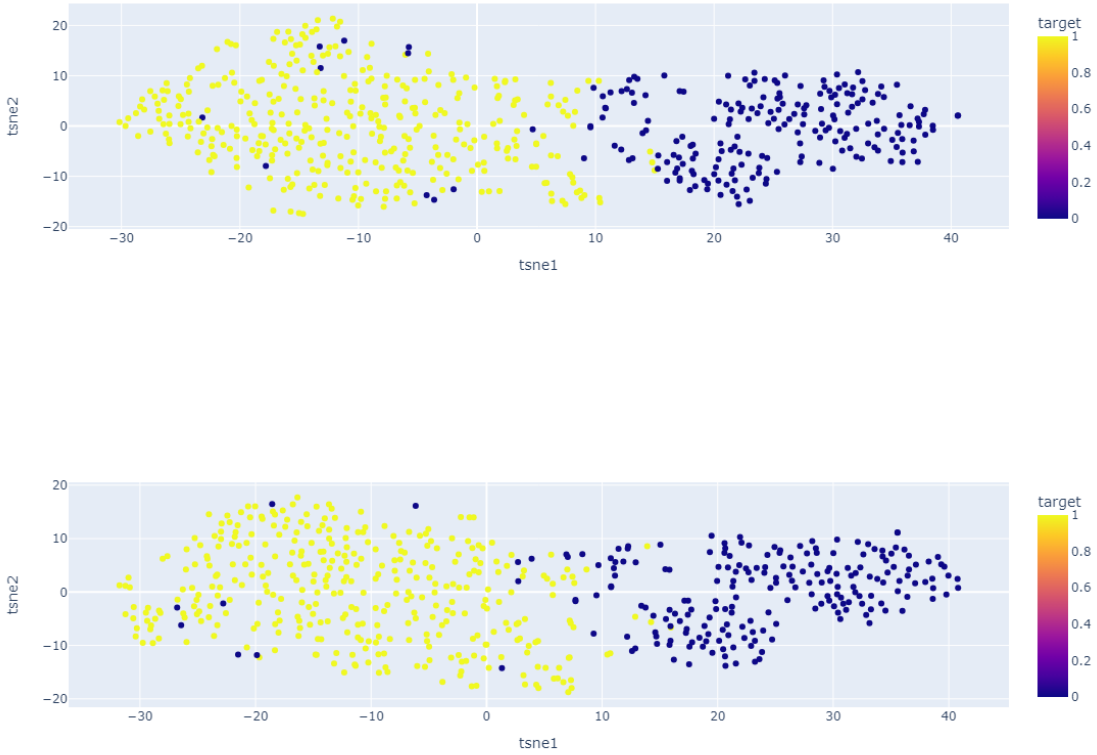
[9]:
```
scaler = sklearn.preprocessing.StandardScaler()
X_std = scaler.fit_transform(X)

for rs in range(0,5):
```

```
    # Create TSNE instance with n_components=2 and random_state=rs
    tsne = TSNE(n_components=2, random_state=rs)
    X_tsne = tsne.fit_transform(X_std)
    df = pd.DataFrame(data=np.c_[X_tsne, y], columns=['tsne1', 'tsne2',␣
↪'target'])
    # Plot dataframe using plotly.express.scatter
    fig = px.scatter(df, x='tsne1', y='tsne2', color=df.target)
    fig.show()
```









EXPLANATION:

t-SNE is a technique used for reducing the dimensionality of data while preserving its structure and relationships. We are iterate through different random states to observe the variability in t-SNE projections. Using Plotly Express (px.scatter), we create a scatter plot of the t-SNE projection. The 'tsne1' and 'tsne2' columns are used as the x and y axes, and the 'target' column is used for color-coding points. We are ploting the t-SNE projections using Plotly Express for each random state to observe how the data is clustered and distributed in 2D space. It's useful for visualizing high-dimensional data and exploring its structure.

Insights from the scatterplots: 1. Data points within the same cluster are more similar to each other in the original high-dimensional space as differenciated by the colour. 2. The clusters seems well seperated, this indicates that the classes are distinct in the original feature space. 3. Since t-SNE uses randomness in the optimization process, running t-SNE with different random seeds shows stability of the clusters. 4. Some points are overlapping with data points from other cluster. It's possible that the overlapping data points represent cases that are ambiguous or challenging to classify correctly. In a classification task, these instances might be misclassified due to their similarity to multiple classes.

REFERENCES:

[7] K. E. (2020, April 13). t-SNE clearly explained. Towards Data Science. https://towardsdatascience.com/t-sne-clearly-explained-d84c537f53a

[ ]: