```
Running perceptron on the airline satisfaction dataset and experimenting with kernel methods
using an implementation of kernelized perceptron from scratch and kernel approximation.
@arashsm79
```

Contents:

In [1]:
```python
import os
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Pre-processing

Before creating a model from the given data, I first have to clean it and get it ready for further processing in the perceptron. I will explain each step one by one.

In [2]:
```python
# Import the raw data as a pandas data frame
raw_train_data = pd.read_csv('train.csv')
```

In [3]:
```python
raw_train_data.head()
```

Out[3]:

	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	Bag han
0	0	70172	Male	Loyal Customer	13	Personal Travel	Eco Plus	460	3	4	...	5	4	3	
1	1	5047	Male	disloyal Customer	25	Business travel	Business	235	3	2	...	1	1	5	
2	2	110028	Female	Loyal Customer	26	Business travel	Business	1142	2	2	...	5	4	3	
3	3	24026	Female	Loyal Customer	25	Business travel	Business	562	2	5	...	2	2	5	
4	4	119299	Male	Loyal Customer	61	Business travel	Business	214	3	3	...	3	3	4	

5 rows × 25 columns

In [4]:
```python
raw_train_data.shape
```

Out[4]:
```
(103904, 25)
```

In [5]:
```python
raw_train_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 103904 entries, 0 to 103903
Data columns (total 25 columns):
 #   Column                             Non-Null Count   Dtype  
---  ------                             --------------   -----  
 0   Unnamed: 0                         103904 non-null  int64  
 1   id                                 103904 non-null  int64  
 2   Gender                             103904 non-null  object 
 3   Customer Type                      103904 non-null  object 
 4   Age                                103904 non-null  int64  
 5   Type of Travel                     103904 non-null  object 
 6   Class                              103904 non-null  object 
 7   Flight Distance                    103904 non-null  int64  
 8   Inflight wifi service              103904 non-null  int64  
 9   Departure/Arrival time convenient  103904 non-null  int64  
 10  Ease of Online booking             103904 non-null  int64  
 11  Gate location                      103904 non-null  int64  
 12  Food and drink                     103904 non-null  int64  
 13  Online boarding                    103904 non-null  int64  
 14  Seat comfort                       103904 non-null  int64  
 15  Inflight entertainment             103904 non-null  int64  
 16  On-board service                   103904 non-null  int64  
 17  Leg room service                   103904 non-null  int64  
 18  Baggage handling                   103904 non-null  int64  
 19  Checkin service                    103904 non-null  int64  
 20  Inflight service                   103904 non-null  int64  
 21  Cleanliness                        103904 non-null  int64  
 22  Departure Delay in Minutes         103904 non-null  int64  
 23  Arrival Delay in Minutes           103594 non-null  float64
 24  satisfaction                       103904 non-null  object 
dtypes: float64(1), int64(19), object(5)
memory usage: 19.8+ MB
```

First we need to seperate the useless columns from the actual data. The first and the second columns seem to have no information of consequence, thus, I will remove them. I also need to seperate the labels from the actual data.

In [6]:
```
train_data = raw_train_data[raw_train_data.columns[2:-1]].copy()
train_data_label = raw_train_data['satisfaction']
```

From the above information, we can see that 'Arrival Delay in Minutes' has some null values; I will have to replace them with a proper value. (here I chose the mean value)

In [7]:
```
train_data['Arrival Delay in Minutes'].replace(np.nan, train_data['Arrival Delay in Minutes'].mean(),inplace=True)
```

Normalize numerical data into a range from 0 to 1.

Here I have used MinMaxScale which transforms features by scaling each feature to a given range. This estimator scales and translates each feature individually such that it is in the given range on the training set, e.g. between zero and one. The transformation is given by:

```
X_std = (X - X.min(axis=0)) / (X.max(axis=0) - X.min(axis=0))
X_scaled = X_std * (max - min) + min
```

Advantages of normalizing the data:

- speeds up learning and leads to faster convergence
- changes the values of numeric columns in the dataset to a common scale
- helps linear seoerators such as SVMs and perceptrons

In [8]:
```
from sklearn.preprocessing import MinMaxScaler

num_attributes = train_data.select_dtypes(include=['float64', 'int64']).columns.tolist()
minmax_sc = MinMaxScaler()
minmax_sc.fit(train_data[num_attributes])
train_data[num_attributes] = minmax_sc.transform(train_data[num_attributes])
```

It is also possible to use other scalers such as StandardScalar.

In [9]:
```
# from sklearn.preprocessing import StandardScaler
# sc = StandardScaler()
# train_data[num_attributes] = sc.fit_transform(train_data[num_attributes])
```

I need to some how encode categorical data for further processing. Here I have chosen OneHotEncoder, because it seems to be the standard for linear models such as SVM and perceptron.

The features are encoded using a one-hot encoding scheme. This creates a binary column for each category and returns a sparse matrix or dense array

By default, the encoder derives the categories based on the unique values in each feature.

This encoding is needed for feeding categorical data to many scikit-learn estimators, notably linear models and SVMs with the standard kernels.

In [10]:
```
cat_attributes = train_data.select_dtypes(include=['object']).columns.tolist()
train_data = pd.get_dummies(train_data, columns=cat_attributes)
```

In [11]:
```
train_data.head()
```

Out[11]:

	Age	Flight Distance	Inflight wifi service	Departure/Arrival time convenient	Ease of Online booking	Gate location	Food and drink	Online boarding	Seat comfort	Inflight entertainment	...	Arrival Delay in Minutes	Gender_Female	Gen
0	0.076923	0.086632	0.6	0.8	0.6	0.2	1.0	0.6	1.0	1.0	...	0.011364	0	
1	0.230769	0.041195	0.6	0.4	0.6	0.6	0.2	0.6	0.2	0.2	...	0.003788	0	
2	0.243590	0.224354	0.4	0.4	0.4	0.4	1.0	1.0	1.0	1.0	...	0.000000	1	
3	0.230769	0.107229	0.4	1.0	1.0	1.0	0.4	0.4	0.4	0.4	...	0.005682	1	
4	0.692308	0.036955	0.6	0.6	0.6	0.6	0.8	1.0	1.0	0.6	...	0.000000	0	

5 rows × 27 columns

Now we repeat the above steps for test data.

In [12]:
```
# load
raw_test_data = pd.read_csv('test.csv')
test_data = raw_test_data[raw_test_data.columns[2:-1]].copy()
test_data_label = raw_test_data['satisfaction']

# remove nan
test_data['Arrival Delay in Minutes'].replace(np.nan, test_data['Arrival Delay in Minutes'].mean(),inplace=True)

# normalize
test_num_attributes = test_data.select_dtypes(include=['float64', 'int64']).columns.tolist()
minmax_sc = MinMaxScaler()
minmax_sc.fit(test_data[test_num_attributes])
test_data[test_num_attributes] = minmax_sc.transform(test_data[test_num_attributes])

# encode categorical data
test_cat_attributes = test_data.select_dtypes(include=['object']).columns.tolist()
test_data = pd.get_dummies(test_data, columns=test_cat_attributes)
```

In [13]:
```
test_data.head()
```

Out[13]:

	Age	Flight Distance	Inflight wifi service	Departure/Arrival time convenient	Ease of Online booking	Gate location	Food and drink	Online boarding	Seat comfort	Inflight entertainment	...	Arrival Delay in Minutes	Gender_Female	Gen
0	0.576923	0.026050	1.0	0.8	0.6	0.75	0.6	0.8	0.50	1.0	...	0.039462	1	
1	0.371795	0.571890	0.2	0.2	0.6	0.00	1.0	0.8	1.00	0.8	...	0.000000	1	
2	0.166667	0.032512	0.4	0.0	0.4	0.75	0.4	0.4	0.25	0.4	...	0.000000	0	
3	0.474359	0.675687	0.0	0.0	0.0	0.25	0.6	0.8	0.75	0.2	...	0.005381	0	
4	0.538462	0.232431	0.4	0.6	0.8	0.50	0.8	0.2	0.25	0.4	...	0.017937	1	

5 rows × 27 columns

1-Perceptron

In [14]:
```
from sklearn.linear_model import Perceptron
```

In [15]:
```
p = Perceptron()
p.fit(train_data, train_data_label)
```

Out[15]:
```
Perceptron()
```

In [16]:
```
from sklearn.metrics import accuracy_score

# predict on train
pred_train = p.predict(train_data)
train_score = accuracy_score(train_data_label, pred_train)
print("Train data accuracy: ", "{:.2f}%".format(train_score*100))

# predict on test
pred_test = p.predict(test_data)
test_score = accuracy_score(test_data_label, pred_test)
print("Test data accuracy: ", "{:.2f}%".format(test_score*100))
```

```
Train data accuracy:  82.11%
Test data accuracy:  82.78%
```

2-Non-linear Perceptron

Kernel Trick

There are no standard Perceptron implementations that use the kernel trick. Thus, I have to implement a binary classifier perceptron from scratch.

I have used the algorithm described in the following resources:

- Shawe-Taylor, John; Cristianini, Nello (2004). Kernel Methods for Pattern Analysis. Cambridge University Press. pp. 241–242.
- https://en.wikipedia.org/wiki/Kernel_perceptron
- https://webpages.charlotte.edu/rbunescu/courses/ou/ml4900/lecture06.pdf
- https://alex.smola.org/teaching/pune2007/pune_3.pdf

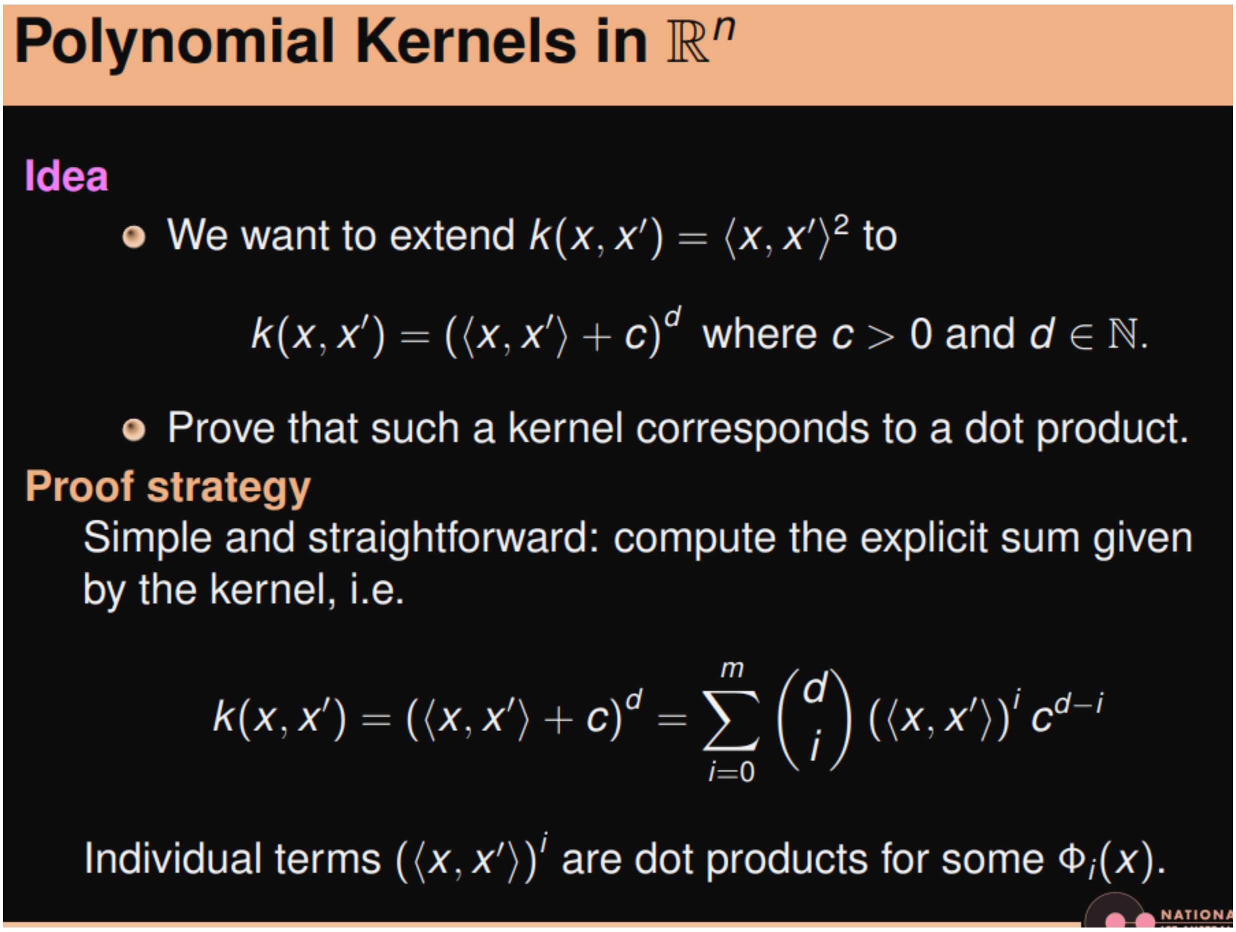
The gist of it is that, we form a dual problem and create a kernelized perceptron algorithm that uses the dot products of the training samples. Then we use the `kernel trick` to compute the dot product in the higher dimension using low dimension data.

Since this is a binary classification (satisfied or dissatisfied/neutral, we don't need to worry about handling multiple classes using methods such as one-vs-one or one-vs-all).

This code this written in python (in contrast to library functions of sklearn that are implemented in C) and thus, using the whole train data takes an infeasable amount of time.

In [49]:
```
reduced_train_data = train_data.head(1000).copy()
reduced_train_data_label = train_data_label.head(1000).copy()
reduced_train_data_label = pd.get_dummies(reduced_train_data_label, columns='satisfaction', drop_first=True)

reduced_test_data = test_data.head(1000).copy()
reduced_test_data_label = test_data_label.head(1000).copy()
reduced_test_data_label = pd.get_dummies(reduced_test_data_label, columns='satisfaction', drop_first=True)
```

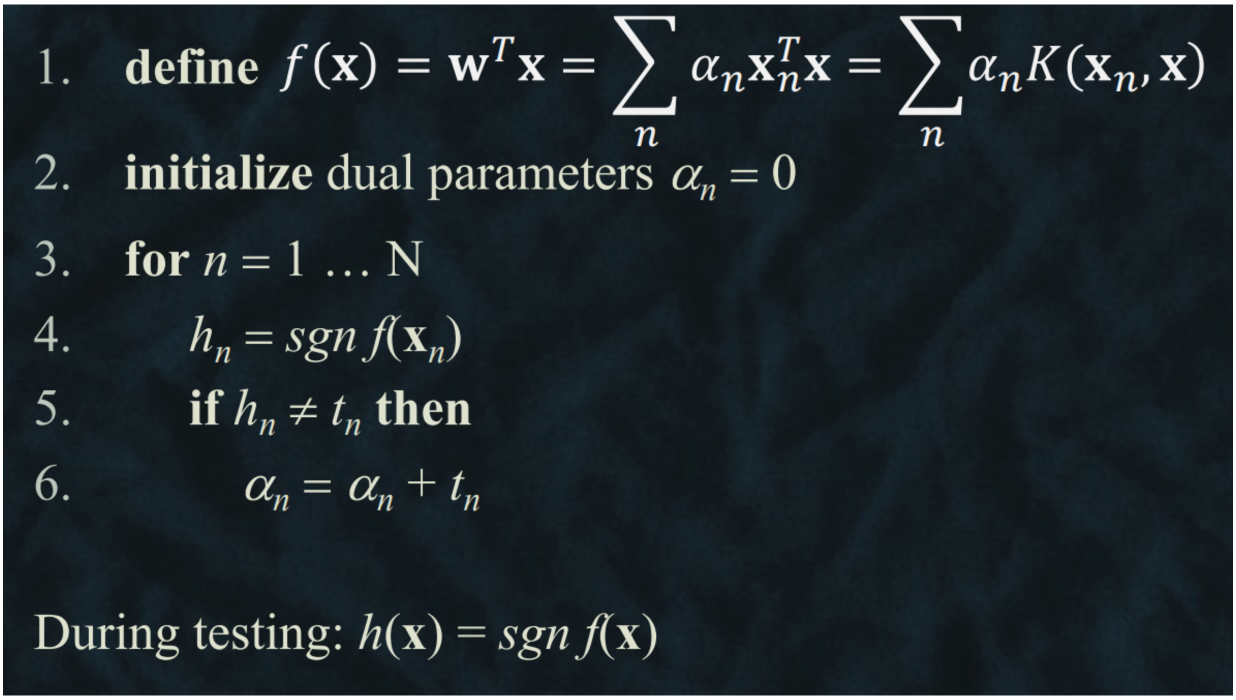


In [43]:
```
def polynomial_kernel(X1, X2, c=1, d=2):
    return (np.dot(X1, X2) + c) ** d
```

In [44]:
```
alpha = []
support_vectors = []
support_vectors_y = []
epsilon = 1e-10
T = 2
kernel = polynomial_kernel
```

In [45]:
```
# generate the gram matrix
def gram_matrix(X1, X2 = None):
    if X2 is None:
        X2 = X1
```

```python
    n, _ = X1.shape
    m, _ = X2.shape
    K = np.zeros((n, m))
    for i in range(n):
        for j in range(m):
            K[i, j] = kernel(X1[i], X2[j])
    return K
```



In [46]:
```python
def fit(train_data_x, train_data_y):
    global alpha
    global support_vectors
    global support_vectors_y
    n, dim = train_data_x.shape
    alpha = np.zeros(n, dtype=np.float64)

    K = gram_matrix(train_data_x)

    for t in range(T):
        for i in range(n):
            if np.sign(np.sum(alpha * train_data_y * K[:, i])) != train_data_y[i]:
                alpha[i] += 1

    support_vectors_mask = alpha > epsilon
    alpha = alpha[support_vectors_mask]
    support_vectors = train_data_x[support_vectors_mask]
    support_vectors_y = train_data_y[support_vectors_mask]
```

In [47]:
```python
def predict(X):
    y_predict = np.zeros(len(X))
    global alpha
    global support_vectors
    global support_vectors_y
    for i in range(len(X)):
        sum = 0
        for j in range(len(alpha)):
            curr_alpha = alpha[j]
            support_vector_y = support_vectors_y[j]
            support_vector = support_vectors[j]
            sum += curr_alpha * support_vector_y * \
                kernel(X[i], support_vector)
        y_predict[i] = sum
    return np.sign(y_predict)
```

In [50]:
```python
fit(reduced_train_data.to_numpy(), reduced_train_data_label.to_numpy())
pred_test = predict(reduced_test_data.to_numpy())
test_score = accuracy_score(reduced_test_data_label, pred_test)
print("Test data accuracy: ", "{:.2f}%".format(test_score*100))
```

```
Test data accuracy:  45.50%
```

the above score is only for 1000 instances of data! for the whole set, it will be much higher.

Kernel Approximation

Here is what I learned from reading the sklearn documentations about this approach:

Kernel approximation performs non-linear transformations of the input, which can serve as a basis for linear classification or other algorithms.

The advantages of using Kernel approximatio compared to the kernel trick:

- can be better suited for online learning
- can significantly reduce the cost of learning with very large datasets
- Standard kernelized SVMs do not scale well to large datasets, but using an approximate kernel map it is possible to use much more efficient linear SVMs.

2 methods are described in the sklearn documentations:

- Method for Kernel Approximation
- RadiaNystroeml Basis Function Kernel

RBF

In [82]:
```
from sklearn import pipeline
from sklearn.kernel_approximation import RBFSampler

# creating a pipeline
rbf_map = RBFSampler(gamma=0.04, random_state=42)
rbf_approx_perceptron = pipeline.Pipeline([("rbf", rbf_map), ("Perceptron", Perceptron())])

# fitting on train data
rbf_approx_perceptron.fit(train_data, train_data_label)

# predict test and compute score
pred_test = rbf_approx_perceptron.predict(test_data)
test_score = accuracy_score(test_data_label, pred_test)
print("Test data accuracy: ", "{:.2f}%".format(test_score*100))
```

```
Test data accuracy:  90.56%
```

Nystroem

In [72]:
```
from sklearn.kernel_approximation import Nystroem

nystroem_map = Nystroem(gamma=0.1, random_state=42)
nystroem_approx_perceptron = pipeline.Pipeline([("nystroem", nystroem_map), ("Perceptron", Perceptron())])

# fitting on train data
nystroem_approx_perceptron.fit(train_data, train_data_label)

# predict test and compute score
pred_test = nystroem_approx_perceptron.predict(test_data)
test_score = accuracy_score(test_data_label, pred_test)
print("Test data accuracy: ", "{:.2f}%".format(test_score*100))
```

```
Test data accuracy:  91.37%
```