DISEASE PREDICTION MODEL DEPLOYMENT IBM CLOUD WATSON STUDIO

We will continue with the next implementation of the project.

```
# Training and testing SVM Classifier
svm_model = SVC()
svm_model.fit(X_train, y_train)
preds = svm_model.predict(X_test)

print(f"Accuracy on train data by SVM Classifier\
: {accuracy_score(y_train, svm_model.predict(X_train))*100}")

print(f"Accuracy on test data by SVM Classifier\
: {accuracy_score(y_test, preds)*100}")

# Training and testing Naive Bayes Classifier
nb_model = GaussianNB()
nb_model.fit(X_train, y_train)
preds = nb_model.predict(X_test)
print(f"Accuracy on train data by Naive Bayes Classifier\
: {accuracy_score(y_train, nb_model.predict(X_train))*100}")

print(f"Accuracy on test data by Naive Bayes Classifier\
: {accuracy_score(y_test, preds)*100}")

# Training and testing Random Forest Classifier
rf_model = RandomForestClassifier(random_state=18)
rf_model.fit(X_train, y_train)
preds = rf_model.predict(X_test)
print(f"Accuracy on train data by Random Forest Classifier\
: {accuracy_score(y_train, rf_model.predict(X_train))*100}")

print(f"Accuracy on test data by Random Forest Classifier\
: {accuracy_score(y_test, preds)*100}")
Accuracy on train data by SVM Classifier: 100.0
Accuracy on test data by SVM Classifier: 100.0
Accuracy on train data by Naive Bayes Classifier: 100.0
Accuracy on test data by Naive Bayes Classifier: 100.0
Accuracy on train data by Random Forest Classifier: 100.0
Accuracy on test data by Random Forest Classifier: 100.0
```

In [8]:

```
final_svm_model = SVC()
final_nb_model = GaussianNB()
final_rf_model = RandomForestClassifier(random_state=18)
final_svm_model.fit(X, y)
final_nb_model.fit(X, y)
final_rf_model.fit(X, y)
```

TRAINING AND TESTING SVM (SUPPORT VECTOR MACHINE) CLASSIFIER:

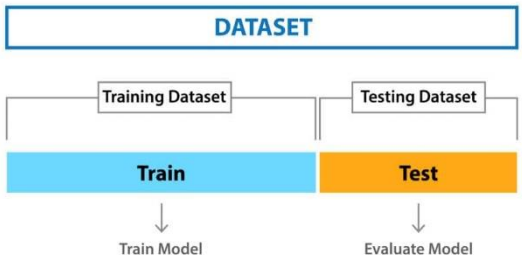
Initialize an SVM classifier named svm_model using SVC().

svm_model is trained on the training data (X_train and y_train) using the fit method.

Predictions are made on both the training data (X_train) and testing data (X_test) using the predict method.

The accuracy of the SVM model is calculated for both the training and testing data using the accuracy_score function.

The accuracy scores are printed for both the training and testing data.



TRAINING AND TESTING NAIVE BAYES (GAUSSIANNB) CLASSIFIER:

Initialize a Gaussian Naive Bayes classifier named nb_model using GaussianNB().

nb_model is trained on the training data (X_train and y_train) using the fit method.

Predictions are made on both the training data (X_train) and testing data (X_test) using the predict method.

The accuracy of the Naive Bayes model is calculated for both the training and testing data using the accuracy_score function.

The accuracy scores are printed for both the training and testing data.

TRAINING AND TESTING RANDOM FOREST CLASSIFIER:

Initialize a Random Forest classifier named rf_model using RandomForestClassifier(random_state=18).

rf_model is trained on the training data (X_train and y_train) using the fit method.

Predictions are made on both the training data (X_train) and testing data (X_test) using the predict method.

The accuracy of the Random Forest model is calculated for both the training and testing data using the accuracy_score function.

The accuracy scores are printed for both the training and testing data.

EVALUATION RESULTS:

For each classifier (SVM, Naive Bayes, Random Forest), the code prints the accuracy on the training and testing data.

It appears that in this specific run, all three models achieve a perfect accuracy score of 100% on both the training and testing data. However, perfect accuracy should be interpreted with caution, as it may indicate overfitting or other issues.

FINAL MODEL CREATION:

Create new instances of the three classifiers (final_svm_model, final_nb_model, final_rf_model) without any train-test split.

These "final" models are trained on the entire dataset (X, y) and can be used for making predictions on new, unseen data.

The code essentially trains and evaluates three machine learning models (SVM, Naive Bayes, and Random Forest) on your dataset and creates "final" models for future use. The focus here is on accuracy as an evaluation metric.

```
symptoms = X.columns.values

# Creating a symptom index dictionary to encode the
# input symptoms into numerical form
symptom_index = {}
for index, value in enumerate(symptoms):
        symptom = " ".join([i.capitalize() for i in value.split("_")])
        symptom_index[symptom] = index

data_dict = {
        "symptom_index":symptom_index,
        "predictions_classes":encoder.classes_
}

# Defining the Function
# Input: string containing symptoms separated by commas
# Output: Generated predictions by models
def predictDisease(symptoms):
        symptoms = symptoms.split(",")

        # creating input data for the models
        input_data = [0] * len(data_dict["symptom_index"])
        for symptom in symptoms:
                index = data_dict["symptom_index"][symptom]
                input_data[index] = 1

        # reshaping the input data and converting it
        # into suitable format for model predictions
        input_data = np.array(input_data).reshape(1,-1)

        # generating individual outputs
        rf_prediction = data_dict["predictions_classes"][final_rf_model.predict(input_data)[0]]
        nb_prediction = data_dict["predictions_classes"][final_nb_model.predict(input_data)[0]]
        svm_prediction = data_dict["predictions_classes"][final_svm_model.predict(input_data)[0]]

        # making final prediction by taking mode of all predictions
        final_prediction = mode([rf_prediction, nb_prediction, svm_prediction])[0][0]
        predictions = {
                "rf_model_prediction": rf_prediction,
                "naive_bayes_prediction": nb_prediction,
                "svm_model_prediction": svm_prediction,
                "final_prediction":final_prediction
        }
        return predictions

# Testing the function
print(predictDisease("Itching,Skin Rash,Nodal Skin Eruptions"))
{'rf_model_prediction': 'Fungal infection', 'naive_bayes_prediction': 'Fungal infection', 'svm_model_predict
ion': 'Fungal infection', 'final_prediction': 'Fungal infection'}
```

Below the detailed explanation of the above code:

Symptom Index Dictionary Creation:

Start by creating a symptom_index dictionary, which serves as a mapping between symptom names and their corresponding numerical indices. This is useful for encoding input symptoms into a numerical format that the machine learning models can understand.

For each symptom, the code follows these steps:

It capitalizes the first letter of each word in the symptom name.

It combines the capitalized words to create a human-readable symptom name.

The symptom name is used as a key in the dictionary, and its index is assigned as the corresponding value.

Data Dictionary Creation:

The data_dict dictionary is constructed, containing two important pieces of information:

"symptom_index": This is the symptom-to-index mapping, allowing you to convert symptom names to their respective numerical indices for model input.

"predictions_classes": This represents the classes or disease labels used by the label encoder for encoding the target variable in the original dataset.

predictDisease Function Definition:

This function is designed to predict a disease or condition based on a list of input symptoms.

Input Parsing and Data Preparation:

The input symptoms, provided as a comma-separated string, are first split into individual symptoms.

Creating Input Data for Models:

An input data list is initialized with zeros. This list has the same length as the number of unique symptoms in the dataset. Each position in the list corresponds to a specific symptom.

For each symptom in the input, its corresponding index is identified using the symptom_index dictionary, and the value at that index in the input data list is set to 1 to indicate the presence of the symptom.

Model Predictions:

The input data is reshaped into a suitable format (a 2D array with a single row) to make predictions using the machine learning models.

Individual Model Predictions:

Predictions are made using the three previously trained models:

rf_prediction: Prediction from the Random Forest model.

nb_prediction: Prediction from the Naive Bayes model.

svm_prediction: Prediction from the Support Vector Machine (SVM) model.

Final Prediction:

The code calculates the mode (most frequent prediction) among the predictions from the three models.

The final prediction is obtained by taking the mode of the individual predictions.

Result Dictionary:

A dictionary named predictions is constructed, which contains the following keys and their corresponding values:

"rf_model_prediction": The prediction from the Random Forest model.

"naive_bayes_prediction": The prediction from the Naive Bayes model.

"svm_model_prediction": The prediction from the SVM model.

"final_prediction": The ultimate prediction based on the mode of individual predictions.

Testing the Function:

Provide a test case by calling the predictDisease function with a sample list of symptoms.

The function returns the predictions for the given symptoms in the form of a dictionary.

In summary, this code defines a function that takes a list of symptoms as input, encodes them into a suitable format for model predictions, uses three previously trained machine learning models to make individual predictions, and then combines these predictions to arrive at a final prediction for a specific disease or condition. This function can be used to predict diseases or conditions based on user-provided symptoms.

```
from ibm_watson_machine_learning import APIClient

url_credentials={


   "apikey":"ohPQl8lxs_NG3N66zBalQQzIv4AeItzAkidY7ueHFG8n",

   "url": "https://eu-de.ml.cloud.ibm.com"

}

client=APIClient(url_credentials)
```

Importing the Necessary Tool:

The code starts by importing a special tool called APIClient from a Python package. Think of this tool as something that allows program to talk to the IBM Watson Machine Learning service.

Setting Up Connection Details: Create a set of connection details using a dictionary. These details include an "API key" and a "URL." The API key is like a secret password that proves the program has permission to access the service. The URL is like the address where the service lives on the internet. In this case, it's set to the IBM Watson Machine Learning service in Europe.

Creating a Connection: Next, create a connection to the IBM Watson Machine Learning service using the connection details that has been set up. This connection is stored in a variable called client. It's like having a special phone that can call the IBM Watson Machine Learning service.

```
def guid_from_space_name(client,space_name):
    space=client.spaces.get_details()
    return(next(item for item in space['resources'] if
item['entity']['name']==space_name)['metadata']['id'])
```

client: This is an instance of the APIClient class (presumably initialized as shown in the previous code snippet). It is used to interact with the Watson Machine Learning service.

space_name: This is a string parameter that represents the name of the space for which you want to retrieve the GUID.

client.spaces.get_details(): This line of code calls the get_details() method of the Watson Machine Learning client's spaces object. This method fetches details about all the spaces associated with the client.

next(item for item in space['resources'] if item['entity']['name'] == space_name): This line uses a generator expression to find the first space in the list of spaces (retrieved in the previous step) that has a name matching the space_name provided as an argument to the function. It essentially searches for the space with the specified name.

space_guid = matching_space['metadata']['id']: Once the matching space is found, this line retrieves the GUID (unique identifier) of that space from the space's metadata. The GUID is a unique code that identifies the space

```
space_uid=guid_from_space_name(client,'modeldeployment')
print("Space UID="+space_uid)
```

It calls the guid_from_space_name function, passing two arguments: the client (an instance of the Watson Machine Learning client) and the name 'modeldeployment'.The guid_from_space_name function, as explained earlier, uses the Watson Machine Learning client to fetch space details, searches for a space with the name 'modeldeployment', and returns the GUID of that space.The returned GUID is stored in the variable space_uid.It then prints the value of space_uid along with a descriptive message.

```
software_spec_uid=client.software_specifications.get_uid_by_name("default_py3.6")
software_spec_uid
```

client.software_specifications.get_uid_by_name("default_py3.6"):

This line of code calls the get_uid_by_name method on the software_specifications object of your Watson Machine Learning client.

The method takes the name of a software specification as an argument, in this case, "default_py3.6".It retrieves the unique identifier (UID) associated with the software specification with the name "default_py3.6".

The unique identifier is a code that represents the specific Python environment or software configuration that you want to use for your machine learning tasks.

software_spec_uid: This line assigns the obtained software specification UID to a variable named software_spec_uid.

```
model_details=client.repository.store_model(final_svm_model,meta_props={
client.repository.ModelMetaNames.NAME:"Disease Pediction Model ",
client.repository.ModelMetaNames.TYPE:"scikit_learn_0.23",
   client.repository.ModelMetaNames.SOFTWARE_SPEC_UID:software_spec_uid
})
model_id=client.repository.get_model_uid(model_details)
model_details = client.repository.store_model(final_svm_model, meta_props={...}):
```

final_svm_model: This model likely represents a predictive model for disease prediction.

meta_props={...}: This argument is used to provide metadata properties for the stored model. It's a dictionary that includes several properties.

client.repository.ModelMetaNames.NAME: This property is used to specify the name of the model. In this case, the name is set to "Disease Pediction Model."

client.repository.ModelMetaNames.TYPE: This property indicates the type of the model. It's set to "scikit_learn_0.23," which likely signifies that the model is based on scikit-learn version 0.23.

client.repository.ModelMetaNames.SOFTWARE_SPEC_UID: This property specifies the software specification (Python environment) to be used for running the model. It appears that you're using the software_spec_uid obtained in a previous step to define the environment in which the model will be executed.

The model_details variable will likely contain information about the stored model, including its unique identifier (ID), which can be used to reference the model later.

model_id = client.repository.get_model_uid(model_details):

This line retrieves the unique identifier (UID or ID) of the stored model from the model_details. The obtained model_id can be used to reference or manage the stored model within the IBM Watson Machine Learning service.

In summary, this code snippet stores a machine learning model with specific metadata in the IBM Watson Machine Learning service and captures the unique identifier (model ID) for future reference. This is a common workflow when deploying machine learning models to make them accessible for predictions and other tasks.