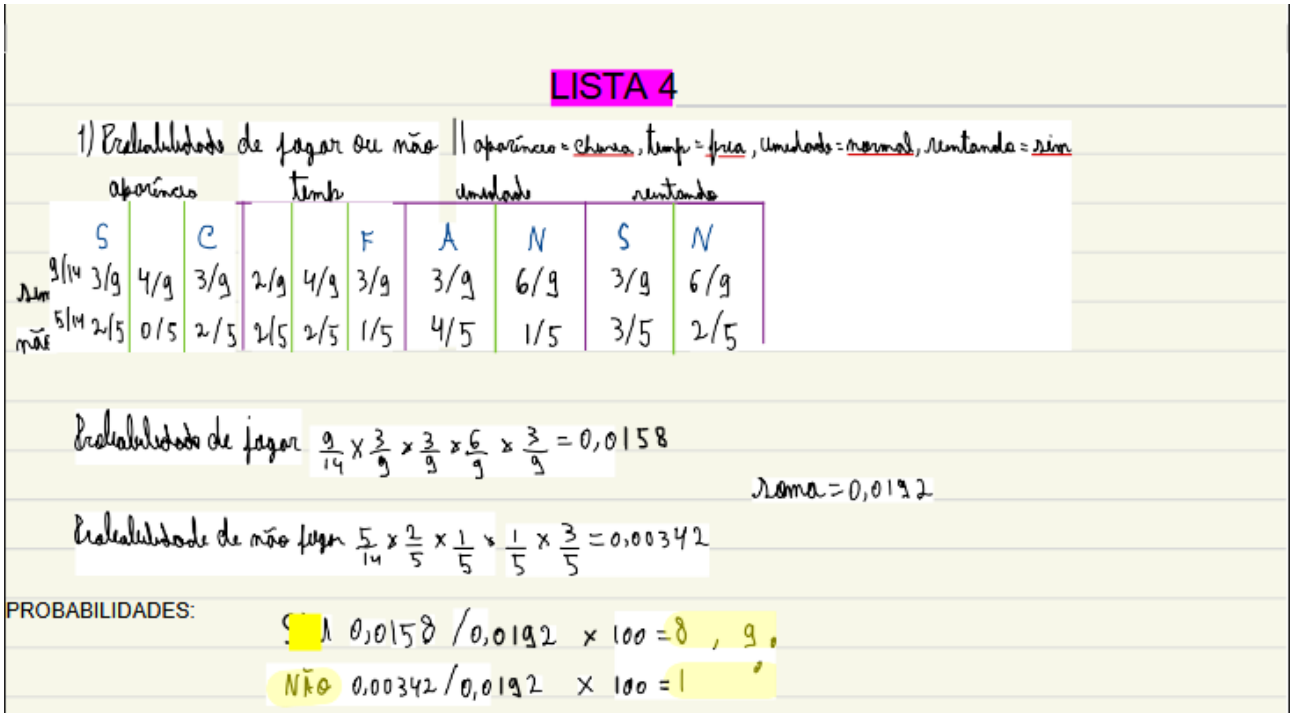


**2)**
```
def naive_bayes_probability(aparencia, temperatura, umidade, ventando):
    # Probabilidades a priori
    prob_jogar = 0.8229  # P(jogar)
    prob_nao_jogar = 0.1771  # P(não jogar)

    # Probabilidades condicionais para jogar
    prob_aparencia_jogar = {
        'sol': 3/9,
        'nublado': 4/9,
        'chuva': 3/9
    }
    prob_temperatura_jogar = {
        'quente': 2/9,
        'agradavel': 4/9,
        'fria': 3/9
    }
    prob_umidade_jogar = {
        'alta': 3/9,
        'normal': 6/9
    }
    prob_ventando_jogar = {
        'sim': 3/9,
        'não': 6/9
    }

    # Probabilidades condicionais para não jogar
    prob_aparencia_nao_jogar = {
        'sol': 2/5,
        'nublado': 0/5,
        'chuva': 2/5
    }
    prob_temperatura_nao_jogar = {
        'quente': 2/5,
        'agradavel': 2/5,
        'fria': 1/5
    }
```

```
    prob_umidade_nao_jogar = {
        'alta': 4/5,
        'normal': 1/5
    }
    prob_ventando_nao_jogar = {
        'sim': 3/5,
        'não': 2/5
    }

    probabilidade_jogar = (
        prob_aparencia_jogar[aparencia]
        * prob_temperatura_jogar[temperatura]
        * prob_umidade_jogar[umidade]
        * prob_ventando_jogar[ventando]
        * prob_jogar
    )

    probabilidade_nao_jogar = (
        prob_aparencia_nao_jogar[aparencia]
        * prob_temperatura_nao_jogar[temperatura]
        * prob_umidade_nao_jogar[umidade]
        * prob_ventando_nao_jogar[ventando]
        * prob_nao_jogar
    )

    # Normalizando as probabilidades para obter a probabilidade posterior
    normalizacao = probabilidade_jogar + probabilidade_nao_jogar

    probabilidade_jogar_posterior = probabilidade_jogar / normalizacao
    probabilidade_nao_jogar_posterior = probabilidade_nao_jogar / normalizacao

    return probabilidade_jogar_posterior, probabilidade_nao_jogar_posterior

# Exemplo de uso:
aparencia = 'sol'
temperatura = 'quente'
umidade = 'alta'
ventando = 'sim'

prob_jogar, prob_nao_jogar = naive_bayes_probability(aparencia, temperatura, umidade, ventando)
print(f'Probabilidade de jogar: {prob_jogar:.4f}')
print(f'Probabilidade de não jogar: {prob_nao_jogar:.4f}')
```

### 3) Implementar Random Forest:

```
rf_model = RandomForestClassifier(random_state=42)

# Treinar o modelo
rf_model.fit(X_train, y_train)
```

### Implementar Naive Bayes e Árvore de Decisão:

```
nb_model = GaussianNB()

nb_model.fit(X_train, y_train)

dt_model = DecisionTreeClassifier(random_state=42)

dt_model.fit(X_train, y_train)
```

**Comparar os Modelos:**

```
# Prever os resultados nos dados de teste
rf_pred = rf_model.predict(X_test)
nb_pred = nb_model.predict(X_test)
dt_pred = dt_model.predict(X_test)

# Calcular acurácia dos modelos
rf_accuracy = accuracy_score(y_test, rf_pred)
nb_accuracy = accuracy_score(y_test, nb_pred)
dt_accuracy = accuracy_score(y_test, dt_pred)

print(f'Acurácia do Random Forest: {rf_accuracy:.4f}')
print(f'Acurácia do Naive Bayes: {nb_accuracy:.4f}')
print(f'Acurácia da Árvore de Decisão: {dt_accuracy:.4f}')
```

**Ajustar Hiperparâmetros com RandomSearch:**

```
# Definir os hiperparâmetros a serem testados
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

rf_random = RandomizedSearchCV(estimator=rf_model, param_distributions=param_grid,
n_iter=100, cv=3, verbose=2, random_state=42, n_jobs=-1)

rf_random.fit(X_train, y_train)

print("Melhores hiperparâmetros encontrados:")
print(rf_random.best_params_)

best_rf_model = rf_random.best_estimator_

# Avaliar o modelo ajustado nos dados de teste
best_rf_pred = best_rf_model.predict(X_test)
best_rf_accuracy = accuracy_score(y_test, best_rf_pred)

print(f'Acurácia do Random Forest com melhores hiperparâmetros: {best_rf_accuracy:.4f}')
```

**4)**
Utilizando-se o algoritmo Apriori, um suporte mínimo aceitável de 0.3 e confiança de 0.8, o número de ItensSets 1, 2, 3 e de regras a partir desta base de dados são: 2.

**5)** Após rodar o código, podemos deduzir que: ele calculará as probabilidades de jogar e não jogar com base nas condições específicas de aparencia='sol', temperatura='quente', umidade='alta' e ventando='sim'.

**6)**

```
data = {
```

```
    'Transação': [1, 2, 3, 4, 5],
    'Itens': [['A', 'B', 'D'],
              ['B', 'C', 'E'],
              ['A', 'B', 'C', 'E'],
              ['A', 'E'],
              ['B', 'D']]
}

df = pd.DataFrame(data)

df_encoded = df['Itens'].apply(lambda x: pd.Series([1] * len(x), index=x)).fillna(0)

frequent_itemsets = apriori(df_encoded, min_support=0.4, use_colnames=True)

print("Conjuntos frequentes com seus suportes:")
print(frequent_itemsets)
```

**7)**

```
def naive_bayes_association_rules(aparencia, temperatura, umidade, ventando):
    prob_jogar = 0.8229  # P(jogar)
    prob_nao_jogar = 0.1771  # P(não jogar)

    prob_aparencia_jogar = {
        'sol': 3/9,
        'nublado': 4/9,
        'chuva': 3/9
    }
    prob_temperatura_jogar = {
        'quente': 2/9,
        'agradavel': 4/9,
        'fria': 3/9
    }
    prob_umidade_jogar = {
        'alta': 3/9,
        'normal': 6/9
    }
    prob_ventando_jogar = {
        'sim': 3/9,
        'não': 6/9
    }

    prob_aparencia_nao_jogar = {
        'sol': 2/5,
        'nublado': 0/5,
        'chuva': 2/5
    }
    prob_temperatura_nao_jogar = {
        'quente': 2/5,
        'agradavel': 2/5,
        'fria': 1/5
    }
    prob_umidade_nao_jogar = {
```

```
        'alta': 4/5,
        'normal': 1/5
    }
    prob_ventando_nao_jogar = {
        'sim': 3/5,
        'não': 2/5
    }

    prob_condicional_jogar = (
        prob_aparencia_jogar[aparencia]
        * prob_temperatura_jogar[temperatura]
        * prob_umidade_jogar[umidade]
        * prob_ventando_jogar[ventando]
    )
    prob_condicional_nao_jogar = (
        prob_aparencia_nao_jogar[aparencia]
        * prob_temperatura_nao_jogar[temperatura]
        * prob_umidade_nao_jogar[umidade]
        * prob_ventando_nao_jogar[ventando]
    )

    prob_X = (prob_conditional_jogar * prob_jogar) + (prob_conditional_nao_jogar *
prob_nao_jogar)

    prob_associations = {}
    products = {
        'álcool': prob_aparencia_nao_jogar['sol'] * prob_temperatura_nao_jogar['quente'] *
prob_umidade_nao_jogar['alta'] * prob_ventando_nao_jogar['sim'],
        'detergente': prob_aparencia_nao_jogar['sol'] *
prob_temperatura_nao_jogar['agradavel'] * prob_umidade_nao_jogar['normal'] *
prob_ventando_nao_jogar['não'],
        'arroz': prob_aparencia_nao_jogar['nublado'] *
prob_temperatura_nao_jogar['agradavel'] * prob_umidade_nao_jogar['normal'] *
prob_ventando_nao_jogar['sim']
    }

    for product, prob in products.items():
        prob_associations[product] = prob / prob_X

    return prob_associations

# Exemplo de uso:
aparencia = 'sol'
temperatura = 'quente'
umidade = 'alta'
ventando = 'sim'

associations = naive_bayes_association_rules(aparencia, temperatura, umidade,
ventando)

# Imprimir as regras de associação
for product, prob in associations.items():
```

```
    print(f"Quem não leva {product}, leva: {prob:.4f}")
```

**8) Exemplo das regras:**

```
# Exemplo de dados:
data = {'ID': [1, 2, 3, 4, 5],
        'Items': [['A', 'B', 'D'],
                  ['B', 'C'],
                  ['A', 'C', 'D'],
                  ['A', 'B'],
                  ['B', 'D']]}

df = pd.DataFrame(data)

# Transformar a coluna 'Items' em one-hot encoding
df_encoded = df['Items'].str.join('|').str.get_dummies()

# Aplicar o algoritmo Apriori para encontrar itemsets frequentes
frequent_itemsets = apriori(df_encoded, min_support=0.2, use_colnames=True)

# Gerar regras de associação
rules = association_rules(frequent_itemsets, metric="confidence", min_threshold=0.7)
print(rules)
```