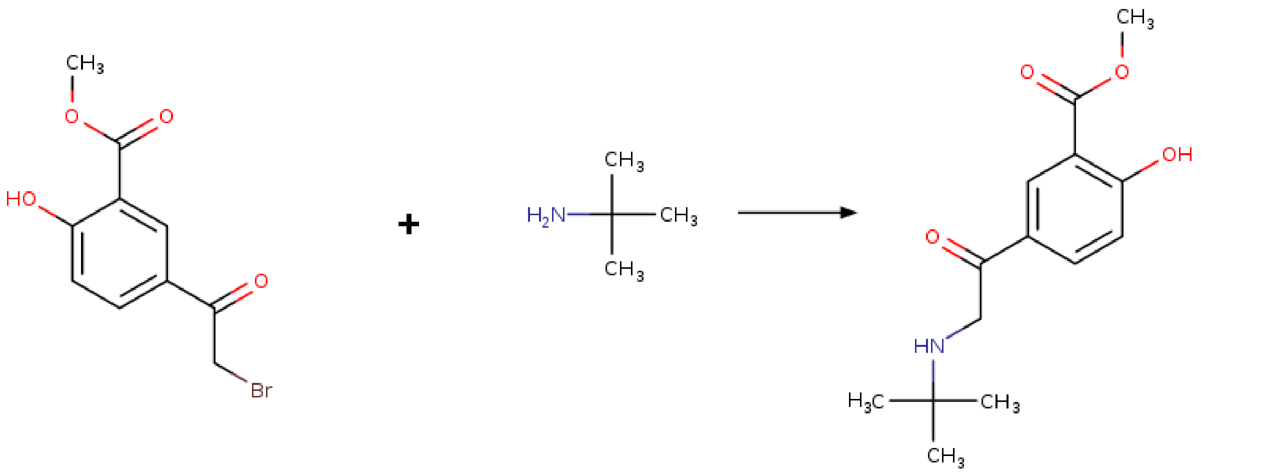
# AI-driven synthesis Pre-Assignment

## 1 Retrosynthesis of (R)-salbutamol

Here is a route to a synthesis the target compound:
In step 1, **Methyl 5-(2-bromoacetyl)-2-hydroxybenzoate (CAS#: 36256-45-8)** and **tert-Butylamine (CAS#: 75-64-9)** are cheap and easy to obtain.

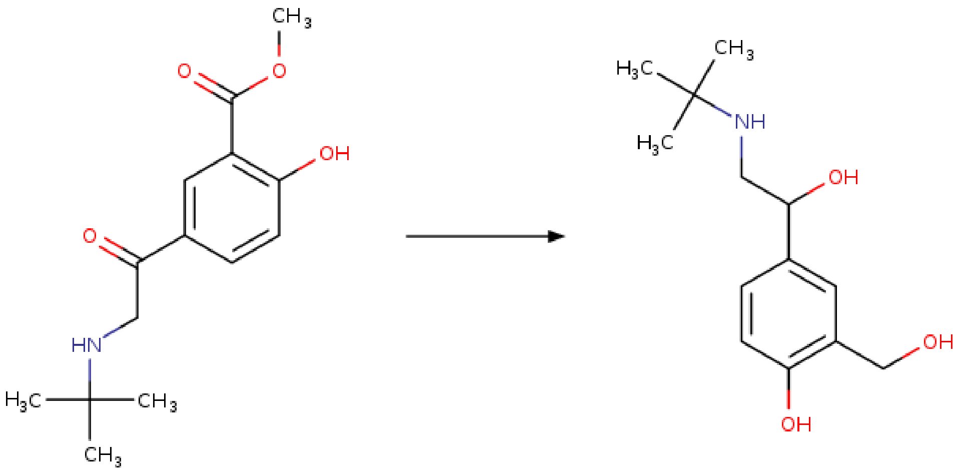
### Step 1:



**Stage #1**: Add *Methyl 5-(2-bromoacetyl)-2-hydroxybenzoate* to *Isopropyl alcohol*, protect with *nitrogen*, cool and add tert-butylamine dropwise.

**Reaction Smiles:** `COC(=O)C1=CC(=CC=C1O)C(=O)CBr.CC(C)(C)N>>COC(=O)C1=C(O)C=CC(=C1)C(=O)CNC(C)(C)C |c:6,8,23,26,28,t:4,lp:1:2,3:2,10:2,12:2,14:3,19:1,21:2,23:2,26:2,32:2,34:1|`

### Step 2:



**Stage #1**: *4-[2-(tert-butylamino)-1-hydroxyethyl]-2-(hydroxymethyl)phenol* (C14H19NO4) With Lithium Aluminium Hydride In Tetrahydrofuran at -10 - 5°C;

**Stage #2:** With *Sodium hydroxide* In water at 0 - 30°C;

Reaction Smiles: `[CH3:18][O:19][C:12](=[O:13])[C:11]1=[C:14]([OH:15])[CH:16]=[CH:17][C:9](=[CH:10]1)[C:7](=[O:8])[CH2:6][NH:5][C:2]([CH3:1])([CH3:4])[CH3:3]>>[CH3:1][C:2]([CH3:3])([CH3:4])[NH:5][CH2:6][CH:7]([OH:8])[C:9]1=[CH:10][C:11]([CH2:12][OH:13])=[C:14]([OH:15])[CH:16]=[CH:17]1 |c:4,7,9,34,t:27,31,lp:1:2,3:2,6:2,12:2,14:1,23:1,26:2,31:2,33:2|`

## 2 Python programming and reaction modeling

*The complete code has been uploaded to GitHub:* cx-luo/rxn-yield-prediction

### 2.1 Data Processing and Plot Scatter

A large amount of data was read from the CSV file, cleaned, standardized, and the data structure for analysis was constructed. Finally, the scatter plot between the reaction temperature and the product yield was plotted.

```python
import pandas as pd
from matplotlib import pyplot as plt

from rxn_logger import logger


class RxnDataProcessing:
    def __init__(self, file_path):
        self.file_path = file_path
        self.results = []
        self.yields_dict = {}
        self.temp_dict = {}

    def csv_data_loader(self, chunk_size=10000):
        """
        Read and process large CSV files in chunks, avoid memory overflow
        """
        for chunk in pd.read_csv(self.file_path, chunksize=chunk_size):
            processed_chunk = self.chunk_standardization(chunk)
            self.create_dicts(processed_chunk)
            self.results.append(processed_chunk)
            logger.info(
                f'f"Finished processing chunk {len(self.results)}, containing
{len(processed_chunk)} rows"')

        final_df = pd.concat(self.results, ignore_index=True)
        return final_df

    @staticmethod
    def chunk_standardization(chunk):
        chunk = chunk[chunk['reaction_SMILES'].notna() & (chunk['reaction_SMILES'] != '')]
        # remove unexpected yield rows
        numeric_fields = ['eqv1', 'eqv2', 'eqv3', 'eqv4', 'eqv5', 'reaction_temperature', 'time',
                          'product1_yield', 'product2_yield']

        # change fields to numeric, none as NaN
        for field in numeric_fields:
            chunk[field] = pd.to_numeric(chunk[field], downcast='float', errors='coerce')

        chunk['reaction_ID'] = pd.to_numeric(chunk['reaction_ID'], downcast='signed',
                                             errors='coerce')

        chunk = chunk[
            (chunk['product1_yield'] >= 0) & (chunk['product1_yield'] <= 100) &
            (chunk['product2_yield'] >= 0) & (chunk['product2_yield'] <= 100)
            ]

        chunk = chunk.dropna(subset=['reaction_ID'])
        # when 'eqv1', 'eqv2', 'eqv3', 'eqv4', 'eqv5', 'reaction_temperature'
        # is none, delete row from chunk
        chunk = chunk.dropna(
            subset=['eqv1', 'eqv2', 'eqv3', 'eqv4', 'eqv5', 'reaction_temperature'])

        return chunk

    def create_dicts(self, chunk):
        # create yields and temperature dicts
        for _, reaction in chunk.iterrows():
            reaction_id = reaction['reaction_ID']
            self.yields_dict[reaction_id] = [reaction['product1_yield'],
reaction['product2_yield']]
            self.temp_dict[reaction_id] = reaction['reaction_temperature']

    def plot_scatter(self):
        # Prepare data: each reaction has two yields, so temperature and yield lists need to be
expanded accordingly
        temperatures = []
        yields = []
        for rid, temperature in self.temp_dict.items():
            reaction_yields = self.yields_dict[rid]
            temperatures.extend([temperature] * len(reaction_yields))
            yields.extend(reaction_yields)

        plt.figure(figsize=(10, 6))
        plt.scatter(temperatures, yields, alpha=0.6, s=50)
        plt.xlabel('Reaction Temperature', fontsize=12)
        plt.ylabel('Product Yield (%)', fontsize=12)
        plt.title('Correlation between Reaction Temperature and Product Yield', fontsize=14)
        plt.grid(True, alpha=0.3)
        plt.tight_layout()
        plt.show()


if __name__ == '__main__':
    rxn_process = RxnDataProcessing('test_organic_reactions.csv')
    df = rxn_process.csv_data_loader()
    print(df.head(10))
    rxn_process.plot_scatter()
```

## 2.2 Build model and  prediction a reaction yield

### 2.2.1 Merge vectors

This method extracts features and labels from the DataFrame, then concats fingerprint vector with extra features (eqv1-5, temperature and time),  finally divides them into training and testing sets, and converts them into tensor format.

```
def gen_train_test_data(self):
    X_list = []
    y_list = []
    for _, row in self.df.iterrows():
        _x = [row['equ1'], row['eqv2'], row['eqv3'], row['eqv4'], row['eqv5'],
              row['reaction_temperature'], row['time']]
        X = np.concatenate((row['fps'], _x))
        y = np.array([row['product1_yield'], row['product2_yield']])
        X_list.append(np.array(X))
        y_list.append(y)

    X_train, X_test, y_train, y_test = train_test_split(
        X_list, y_list, test_size=0.3, random_state=42
    )

    X_train = torch.tensor(X_train, dtype=torch.float32)
    X_test = torch.tensor(X_test, dtype=torch.float32)
    y_train = torch.tensor(y_train, dtype=torch.float32)
    y_test = torch.tensor(y_test, dtype=torch.float32)

    return X_train, X_test, y_train, y_test
```

### 2.2.2 build a simple model

There are to ways for building models:

1. For a single model, the output layer directly outputs two values (multiple outputs).
2. Use two independent models (or two independent output heads) to output each value separately.

Each of these approaches has advantages and disadvantages, which are discussed in more detail below:

**Method 1: Single model, the output layer directly outputs two values**

- Structure: On the last layer of the model, set two nodes. If the two values have different types (such as one classification and one regression), two different output layers may be required (such as one softmax for classification and one linear for regression).
- Advantages: The model shares most feature extraction layers, which can reduce the amount of computation, and the two tasks may promote each other. By sharing the feature extraction layer, the model can learn features that are useful for both tasks.
- Disadvantages: Two tasks may interfere with each other, especially when the correlation between the two tasks is not high. In addition, the balance of the loss function may require careful adjustment.

```
class YieldPredict(nn.Module):
    def __init__(self, input_size=263, hidden_size=128, output_size=2):
        super(YieldPredict, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.fc3 = nn.Linear(hidden_size, output_size)
        self.dropout = nn.Dropout(0.3)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.fc1(x)
        x = self.relu(x)
        x = self.dropout(x)
        x = self.fc2(x)
        x = self.relu(x)
        x = self.dropout(x)
        out = self.fc3(x)
        return out
```

**Method 2: Use two different models (or two independent weight outputs)**

- Structure: Build an independent model for each output value, or set an independent output header for each output value based on a model (that is, fork from a certain intermediate layer, pass through different fully connected layers, etc., and finally output two values).
- Advantages: The two tasks will not interfere with each other and can be optimized independently. This approach may be more effective if the two tasks differ greatly.
- Disadvantages: More parameters are required, the computational volume may be greater, and if the two tasks share features, feature extraction cannot be shared, which may cause the features learned by each model to be less comprehensive than the method.

```
class MultiHeadYieldPredict(nn.Module):
    def __init__(self, input_dim, hidden_dim):
        super().__init__()
        self.shared_backbone = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.ReLU()
        )
        self.head1 = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1)
        )
        self.head2 = nn.Sequential(
            nn.Linear(hidden_dim, hidden_dim // 2),
            nn.ReLU(),
            nn.Linear(hidden_dim // 2, 1)
        )

    def forward(self, x):
        shared_features = self.shared_backbone(x)
        output1 = self.head1(shared_features)
        output2 = self.head2(shared_features)
        return output1, output2
```

### 2.2.3 Train model and Evaluate

These two methods are designed for model training and evaluation.

**Training: save model every 10 epochs.**

```
def train(self, X_train, y_train, num_epochs):
    criterion = nn.MSELoss()
    optimizer = optim.Adam(self.model.parameters(), lr=0.001)

    for epoch in range(num_epochs):
        outputs = model(X_train)
        loss = criterion(outputs, y_train)

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        if (epoch + 1) % 10 == 0:
            torch.save(model.state_dict(), "yield_predictor.pth")
            logger.info(f'Epoch [{epoch + 1}/{num_epochs}], Loss: {loss.item():.4f}')

    return self.model

@torch.no_grad()
def evaluate(self, X_test, y_test):
    criterion = nn.MSELoss()
    outputs = model(X_test)
    _loss = criterion(outputs, y_test)
    logger.info(f'Test Loss: {_loss.item():.4f}')

    predicted_yields = outputs.numpy()
    logger.info(f'Predicted yields: {predicted_yields}')
```

To validate my model, I am actively acquiring real-world datasets. I also intend to compare its performance against established machine learning methods (such as random forest and Bayesian optimization), to provide a comprehensive evaluation of its predictive capabilities.