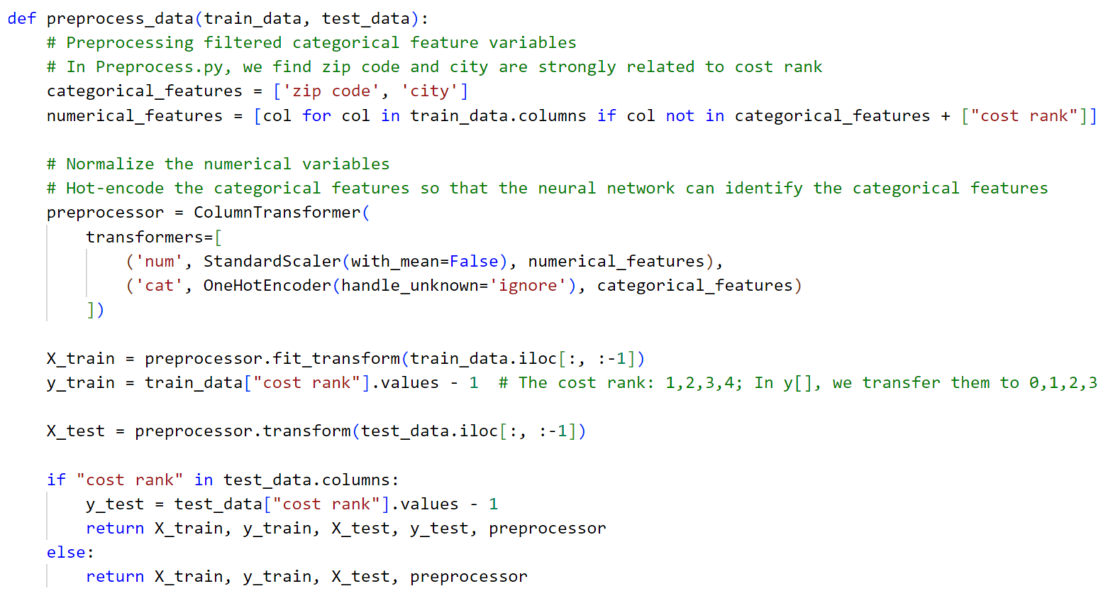
The methodology to optimize the model performance in Task 2

1. Data preprocessing

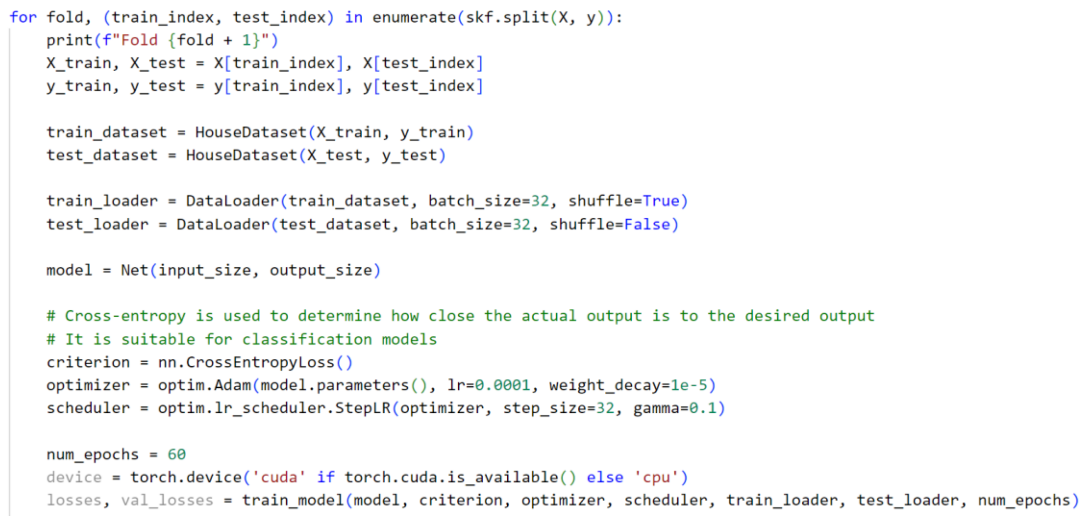
Use “sklearn.ColumnTransformer” to standardize numerical features and one-hot encode categorical features for neural network recognition. (def preprocess_data() in line 75)



```
def preprocess_data(train_data, test_data):
    # Preprocessing filtered categorical feature variables
    # In Preprocess.py, we find zip code and city are strongly related to cost rank
    categorical_features = ['zip code', 'city']
    numerical_features = [col for col in train_data.columns if col not in categorical_features + ["cost rank"]]

    # Normalize the numerical variables
    # Hot-encode the categorical features so that the neural network can identify the categorical features
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', StandardScaler(with_mean=False), numerical_features),
            ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features)
        ])

    X_train = preprocessor.fit_transform(train_data.iloc[:, :-1])
    y_train = train_data["cost rank"].values - 1  # The cost rank: 1,2,3,4; In y[], we transfer them to 0,1,2,3

    X_test = preprocessor.transform(test_data.iloc[:, :-1])

    if "cost rank" in test_data.columns:
        y_test = test_data["cost rank"].values - 1
        return X_train, y_train, X_test, y_test, preprocessor
    else:
        return X_train, y_train, X_test, preprocessor
```

2. Stratified K-fold cross-validation

Use “sklearn.StratifiedKFold” to divide the data set into K mutually exclusive subsets, ensuring that the category distribution of each subset is the same as that of the entire data set. The model is then trained and tested on each subset to evaluate the performance of the model and select the best hyperparameters. (def evaluation() in line 177)



```
for fold, (train_index, test_index) in enumerate(skf.split(X, y)):
    print(f"Fold {fold + 1}")
    X_train, X_test = X[train_index], X[test_index]
    y_train, y_test = y[train_index], y[test_index]

    train_dataset = HouseDataset(X_train, y_train)
    test_dataset = HouseDataset(X_test, y_test)

    train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
    test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)

    model = Net(input_size, output_size)

    # Cross-entropy is used to determine how close the actual output is to the desired output
    # It is suitable for classification models
    criterion = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.0001, weight_decay=1e-5)
    scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=32, gamma=0.1)

    num_epochs = 60
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    losses, val_losses = train_model(model, criterion, optimizer, scheduler, train_loader, test_loader, num_epochs)
```

3. Batch processing

Use “PyTorch.DataLoader” to batch process data to speed up model training.

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32, shuffle=False)
```

4. Batch Normalization

Use BatchNorm1d in each layer of the Net to speed up model training and improve the generalization ability of the model.

5. Dropout regularization

Use Dropout in each layer of the Net to prevent overfitting.

6. L2 regularization

Add L2 regularization term during model training to prevent overfitting.

```
# PyTorch
class Net(nn.Module):
    def __init__(self, input_size, output_size, dropout_prob=0.15, l2_reg=0.00):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(input_size, 256)
        self.bn1 = nn.BatchNorm1d(256)
        self.dropout1 = nn.Dropout(dropout_prob)

        self.fc2 = nn.Linear(256, 128)
        self.bn2 = nn.BatchNorm1d(128)
        self.dropout2 = nn.Dropout(dropout_prob)

        self.fc3 = nn.Linear(128, 64)
        self.bn3 = nn.BatchNorm1d(64)
        self.dropout3 = nn.Dropout(dropout_prob)

        self.fc4 = nn.Linear(64, 32)
        self.bn4 = nn.BatchNorm1d(32)
        self.dropout4 = nn.Dropout(dropout_prob)

        self.fc5 = nn.Linear(32, output_size)

        self.l2_reg = l2_reg ## Adjust the value of l2_reg to find the suitable parameter
    # Function: Relu(), using 0.15 dropout in every layer to reduce fit
    def forward(self, x):
        x = self.dropout1(F.relu(self.bn1(self.fc1(x))))
        x = self.dropout2(F.relu(self.bn2(self.fc2(x))))
        x = self.dropout3(F.relu(self.bn3(self.fc3(x))))
        x = self.dropout4(F.relu(self.bn4(self.fc4(x))))
        x = self.fc5(x)
        return x

    def l2_regularization(self):
        l2_loss = 0
        for param in self.parameters():
            l2_loss += torch.norm(param, p=2) ** 2
        return self.l2_reg * l2_loss
```

7. Learning rate adjustment

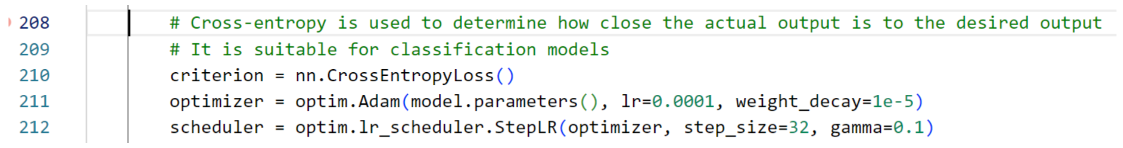
Use “PyTorch.StepLR” to adjust the learning rate. By using this function, you can reduce the learning rate by a fixed multiple, such as 10 times, at the end of each epoch. This method can effectively control the change in the learning rate and improve the performance of the model.

8. Optimizer

The MLP neural network uses the Adam optimizer, which is a gradient-based optimization algorithm that combines the advantages of Adagrad and RMSprop and has good performance and convergence speed. By continuously adjusting model parameters, the Adam optimizer can minimize the loss function and improve model performance.

9. Loss function

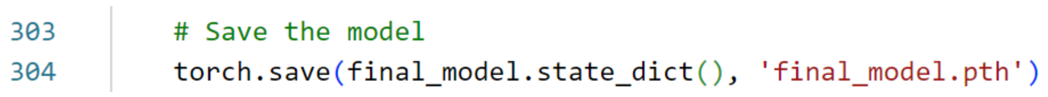
Use the cross-entropy loss function to evaluate the performance of the model. The cross-entropy loss function is very suitable for classification models. It quantifies the gap between the probability distribution of the model output and the real label as a scalar, which is used to optimize the model parameters.



```python
# Cross-entropy is used to determine how close the actual output is to the desired output
# It is suitable for classification models
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.0001, weight_decay=1e-5)
scheduler = optim.lr_scheduler.StepLR(optimizer, step_size=32, gamma=0.1)
```

10. Model saving

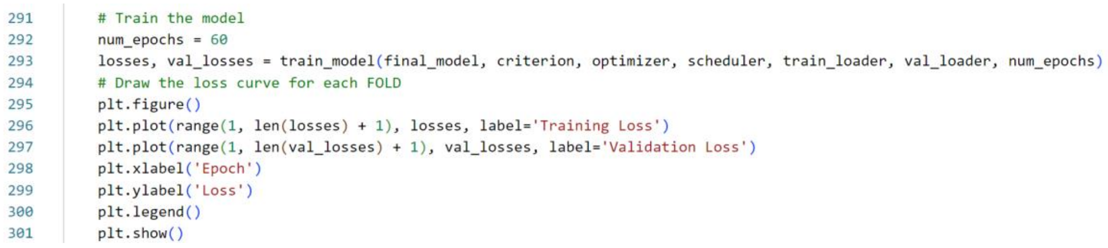
Save method to save the trained model for subsequent use.



```python
# Save the model
torch.save(final_model.state_dict(), 'final_model.pth')
```

11. Visualize the training results

Use Matplotlib to draw the loss function curve during the training process to observe the training effect



```python
# Train the model
num_epochs = 60
losses, val_losses = train_model(final_model, criterion, optimizer, scheduler, train_loader, val_loader, num_epochs)
# Draw the loss curve for each FOLD
plt.figure()
plt.plot(range(1, len(losses) + 1), losses, label='Training Loss')
plt.plot(range(1, len(val_losses) + 1), val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```