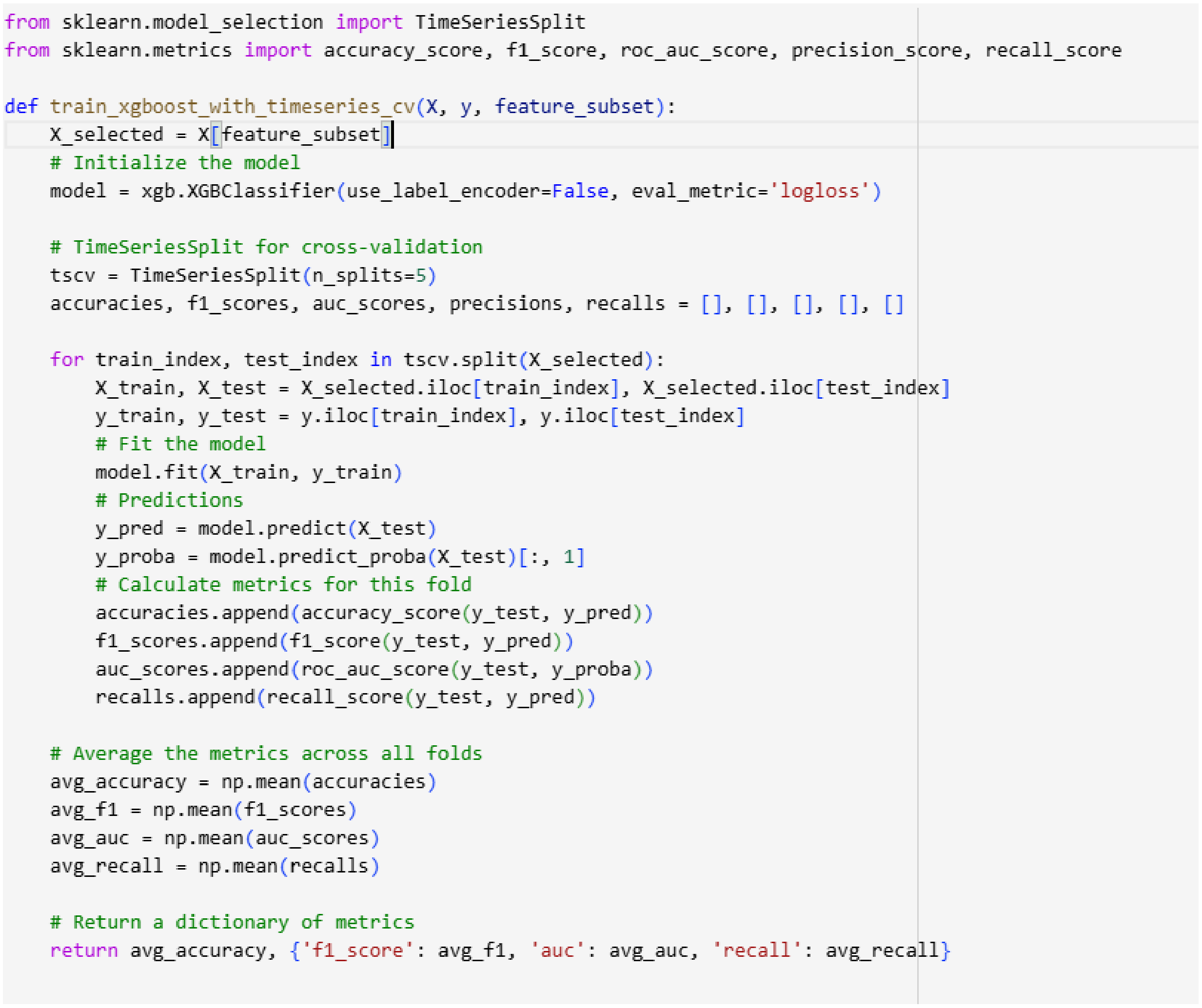
Improving Genetic Algorithm and Model Optimization for XGBoost

1) Improving Diversity in Population Initialization

- Using a broader range in the initial population from a small subset to a large subgroup may help to improve the genetic algorithm.
- Using a broader range for feature subset sizes (e.g., initializing some subsets with three features and others with nine), we can introduce more diversity into the population. This increased variation allows the genetic algorithm to explore more combinations and potentially find a better-performing feature subset.

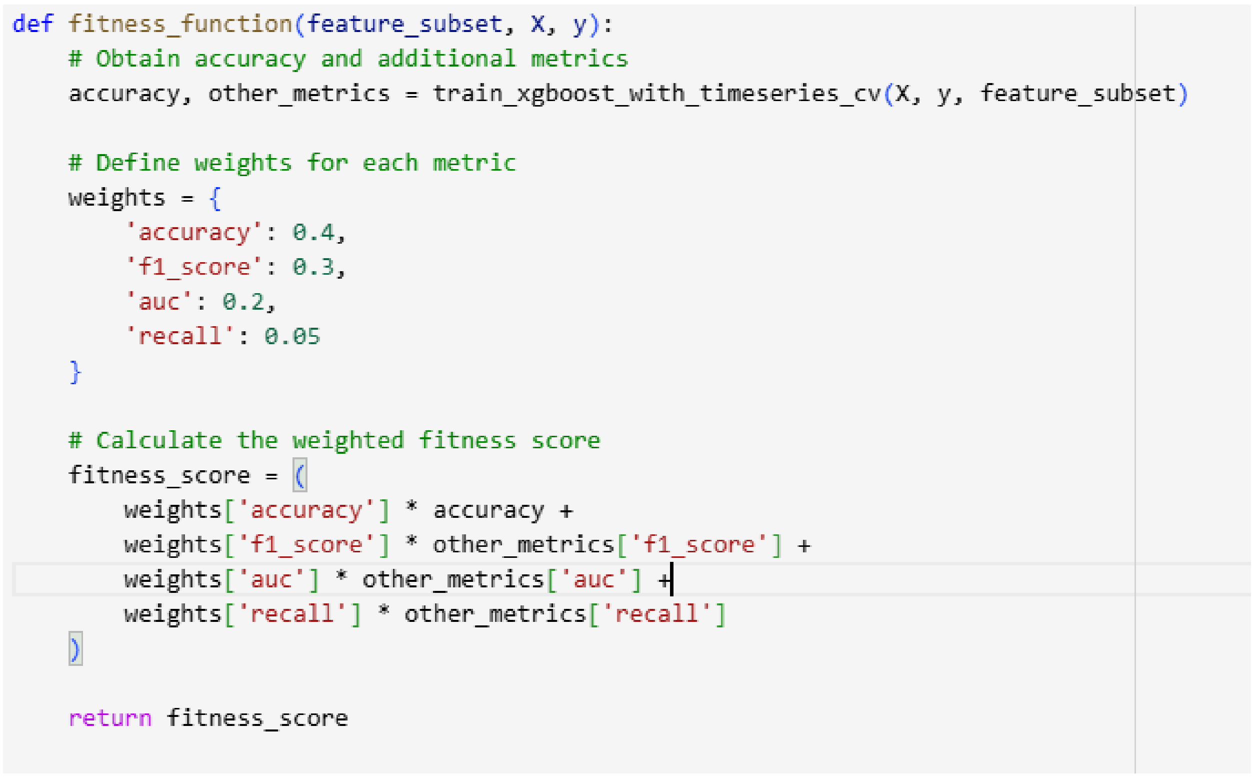
2) Modifying train_xgboost_with_timeseries_cv to return Multiple Metrics

- Currently, it is evaluated just on accuracy.
- We can add F1_score, Area Under Curve (AUC), and Recall to the evaluation process using TimeSeriesSplit for cross-validation. This multi-metric approach provides a more balanced view of model performance
- An example of this can be shown here



```python
from sklearn.model_selection import TimeSeriesSplit
from sklearn.metrics import accuracy_score, f1_score, roc_auc_score, precision_score, recall_score

def train_xgboost_with_timeseries_cv(X, y, feature_subset):
    X_selected = X[feature_subset]
    # Initialize the model
    model = xgb.XGBClassifier(use_label_encoder=False, eval_metric='logloss')

    # TimeSeriesSplit for cross-validation
    tscv = TimeSeriesSplit(n_splits=5)
    accuracies, f1_scores, auc_scores, precisions, recalls = [], [], [], [], []

    for train_index, test_index in tscv.split(X_selected):
        X_train, X_test = X_selected.iloc[train_index], X_selected.iloc[test_index]
        y_train, y_test = y.iloc[train_index], y.iloc[test_index]
        # Fit the model
        model.fit(X_train, y_train)
        # Predictions
        y_pred = model.predict(X_test)
        y_proba = model.predict_proba(X_test)[:, 1]
        # Calculate metrics for this fold
        accuracies.append(accuracy_score(y_test, y_pred))
        f1_scores.append(f1_score(y_test, y_pred))
        auc_scores.append(roc_auc_score(y_test, y_proba))
        recalls.append(recall_score(y_test, y_pred))

    # Average the metrics across all folds
    avg_accuracy = np.mean(accuracies)
    avg_f1 = np.mean(f1_scores)
    avg_auc = np.mean(auc_scores)
    avg_recall = np.mean(recalls)

    # Return a dictionary of metrics
    return avg_accuracy, {'f1_score': avg_f1, 'auc': avg_auc, 'recall': avg_recall}
```

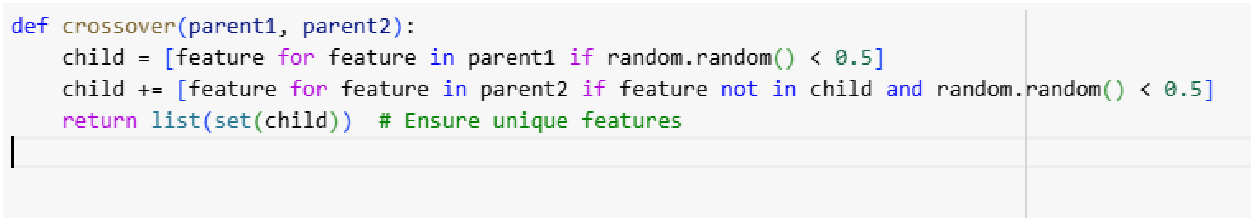
- After calculating multiple metrics, the fitness function combines them with weighted importance. In the fitness Function, we could then weigh the fitness scores like so:



```
def fitness_function(feature_subset, X, y):
    # Obtain accuracy and additional metrics
    accuracy, other_metrics = train_xgboost_with_timeseries_cv(X, y, feature_subset)

    # Define weights for each metric
    weights = {
        'accuracy': 0.4,
        'f1_score': 0.3,
        'auc': 0.2,
        'recall': 0.05
    }

    # Calculate the weighted fitness score
    fitness_score = (
        weights['accuracy'] * accuracy +
        weights['f1_score'] * other_metrics['f1_score'] +
        weights['auc'] * other_metrics['auc'] +
        weights['recall'] * other_metrics['recall']
    )

    return fitness_score
```

3) Refining Crossover Function

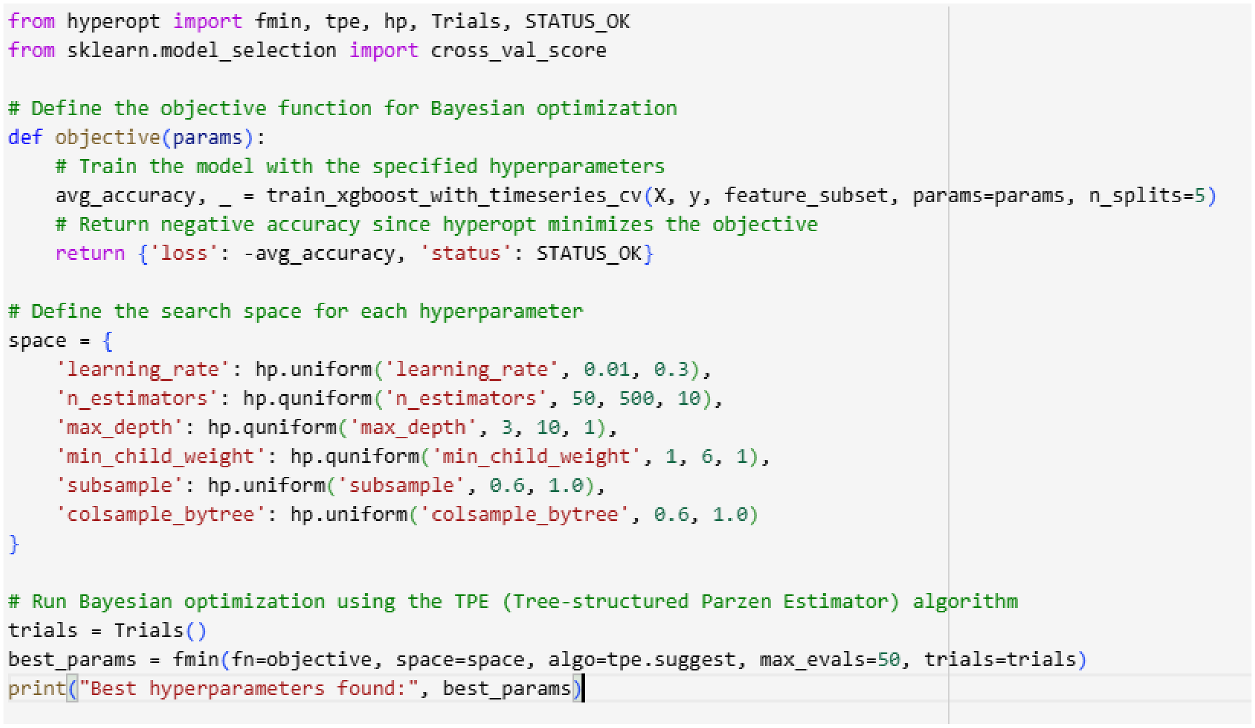
- With our Crossover function, we should try a single point split for crossover with a 50% chance which further increases feature diversity in the child.
- This increases the diversity of child feature subsets.



```
def crossover(parent1, parent2):
    child = [feature for feature in parent1 if random.random() < 0.5]
    child += [feature for feature in parent2 if feature not in child and random.random() < 0.5]
    return list(set(child))  # Ensure unique features
```

4) Adding Bayesian Optimization for Hyperparameter Tuning

- This should control various aspects of the XGBoost model's behavior for how complex the model can be and how aggressive it learns patterns in the data.
- Optimizing these the model can better fit the data and be better generalized to other stocks.
- Basically this will search for the best parameters instead of relying on default values.
- Here is an example of implementation



```python
from hyperopt import fmin, tpe, hp, Trials, STATUS_OK
from sklearn.model_selection import cross_val_score

# Define the objective function for Bayesian optimization
def objective(params):
    # Train the model with the specified hyperparameters
    avg_accuracy, _ = train_xgboost_with_timeseries_cv(X, y, feature_subset, params=params, n_splits=5)
    # Return negative accuracy since hyperopt minimizes the objective
    return {'loss': -avg_accuracy, 'status': STATUS_OK}

# Define the search space for each hyperparameter
space = {
    'learning_rate': hp.uniform('learning_rate', 0.01, 0.3),
    'n_estimators': hp.quniform('n_estimators', 50, 500, 10),
    'max_depth': hp.quniform('max_depth', 3, 10, 1),
    'min_child_weight': hp.quniform('min_child_weight', 1, 6, 1),
    'subsample': hp.uniform('subsample', 0.6, 1.0),
    'colsample_bytree': hp.uniform('colsample_bytree', 0.6, 1.0)
}

# Run Bayesian optimization using the TPE (Tree-structured Parzen Estimator) algorithm
trials = Trials()
best_params = fmin(fn=objective, space=space, algo=tpe.suggest, max_evals=50, trials=trials)
print("Best hyperparameters found:", best_params)
```

- And then pass those best-params into the train_xgboost_with_timeseries_cv.

After my research, I believe these would be the best ways to improve our XGBoost performance.