Untitled-1

November 6, 2024

1 Gradient Boosting for Trees

1.1 (a) Choice of Model and Link Function

(i) Why is Poisson a reasonable choice?

Poisson distribution is a reasonable choice for modeling Y because: 1. Y is in the natural numbers (i.e., it is a count variable), and the Poisson distribution is commonly used for modeling count data. 2. Poisson distribution is suitable when events occur independently, and the expected number of occurrences in a given interval is proportional to some function of covariates X, as represented by $f(X)$.

(ii) Why do we use exp() as our link function?

The exponential function, $\exp(f(X))$, is used as the link function because: 1. It ensures that the expected value $\mathbb{E}[Y|X] = \exp(f(X))$ is always positive, which aligns with the non-negative nature of Poisson-distributed counts. 2. It is a canonical link function for Poisson models, facilitating the application of the generalized linear model framework.

1.2 (b) Loss Function

For Poisson regression, the loss function (negative log-likelihood) in terms of $f(x)$ and Y is:

$$L(f(x), Y) = -Y \cdot f(x) + \exp(f(x))$$

1.3 (c) Gradient of the Loss with respect to $f(x)$

The gradient of the loss function with respect to $f(x)$ is:

$$\frac{dL}{df(x)} = -Y + \exp(f(x))$$

1.4 (d) Choosing Each h_j in First-Order Boosting

In first-order boosting, each $h_j(X)$ is chosen to minimize the loss function at each stage by focusing on the negative gradient of the current model's output. Here's the approach:

1. At each boosting step, we compute the residuals, which for Poisson loss correspond to the difference $Y - \exp(f(X))$.
2. We then fit a regression tree $h_j(X)$ to approximate these residuals.

3. The resulting model is updated by adding $\lambda_j h_j(X)$ to the cumulative model function $f(X)$, weighted by a learning rate λ_j, to optimize convergence.

2 Hyperparameter Selection

Here is a solution draft with explanations for each part:

2.1 (a) Generalization Bound

1. **Setup and Notation**:
   - Let $\hat{R}_{\text{test}}$ denote the empirical risk estimated from the test set.
   - Let R denote the true risk of the model.
   - Let n be the number of training samples, l the number of test samples, and k the number of hyperparameters tried.

2. **Generalization Bound**: Since we are selecting from k different hyperparameter settings, a union bound can be applied over the choices of hyperparameters.

   Using concentration inequalities, such as Hoeffding's inequality, we can bound the deviation between $\hat{R}_{\text{test}}$ and R for a given hyperparameter setting. Specifically, for a given hyperparameter configuration:

$$P\left(|\hat{R}_{\text{test}} - R| \geq \epsilon\right) \leq 2\exp\left(-2l\epsilon^2\right)$$

   Since we are testing k configurations, we extend the bound:

$$P\left(\max_{1\leq j\leq k} |\hat{R}^{(j)}_{\text{test}} - R^{(j)}| \geq \epsilon\right) \leq 2k\exp\left(-2l\epsilon^2\right)$$

   The resulting bound is:

$$P\left(\max_{1\leq j\leq k} |\hat{R}^{(j)}_{\text{test}} - R^{(j)}| < \epsilon\right) \geq 1 - 2k\exp\left(-2l\epsilon^2\right)$$

   This shows that, as l (the number of test samples) grows, the probability that our empirical test error closely approximates the true risk increases. However, a larger k reduces this bound, leading to a looser approximation of the true risk.

2.2 (b) Risks of Using the Test Set for Both Hyperparameter Selection and Generalization Estimation

Using the same test set for hyperparameter selection and to estimate generalization error introduces data leakage, as follows:

1. **Overfitting to the Test Set**: If k is large relative to l, the model is likely to overfit the test set by tailoring hyperparameters to achieve the best performance on the test samples, not on the underlying distribution.

2. **Underestimated Generalization Error**: The generalization error might be underestimated because the model selection process implicitly treats the test set as part of the training process, leading to overly optimistic estimates of performance.

This means that the true risk of the selected model might be higher than what is estimated on the test set.

2.3 (c) Experiment with White Wine Dataset

Let's proceed with a practical application using the white wine dataset. Divide the data randomly into a 70% training set, a 10% validation set, and a 20% test set.

Conclusion: - Based on the MSE values, the `RandomForestRegressor` is the better model for this regression task on the wine quality dataset. It has the lowest test error, indicating stronger predictive performance and generalization. - The difference between validation and test scores is not substantial, showing that the model selection using the validation set was effective without overfitting.

Thus, for this dataset and problem, the `RandomForestRegressor` would be the preferred model.

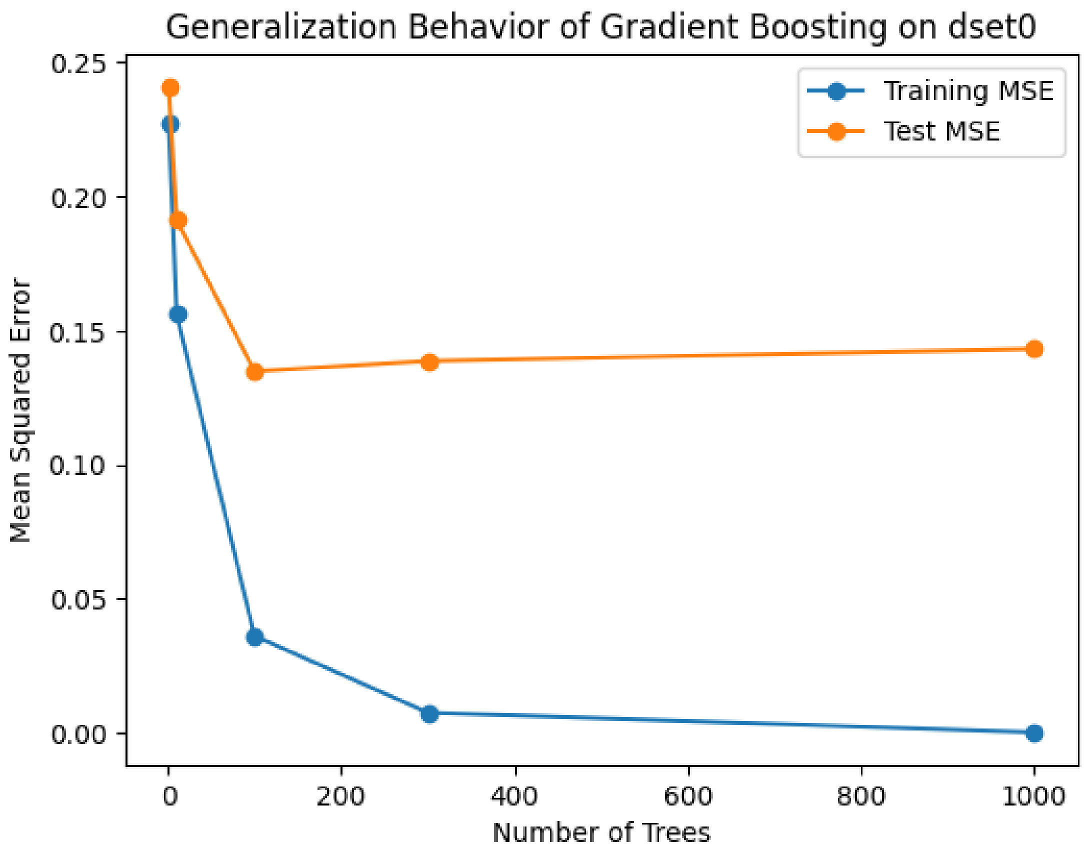
[ ]:
```
import pandas as pd

# Load the uploaded wine dataset
wine_data = pd.read_csv('winequality-white.csv', delimiter=';')

# Display the first few rows of the dataset to understand its structure
wine_data.head()

import pandas as pd
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestRegressor
from sklearn.neighbors import KNeighborsRegressor
from sklearn.metrics import mean_squared_error
import numpy as np

# Load the dataset
wine_data = pd.read_csv('winequality-white.csv', delimiter=';')

# Separate features and target variable
X = wine_data.drop(columns='quality')
y = wine_data['quality']

# Split data into training (70%), validation (10%), and test (20%) sets
X_train, X_temp, y_train, y_temp = train_test_split(X, y, test_size=0.3,␣
↪random_state=42)
X_val, X_test, y_val, y_test = train_test_split(X_temp, y_temp, test_size=2/3,␣
↪random_state=42)  # 10% validation, 20% test

# Define model hyperparameter grids
```

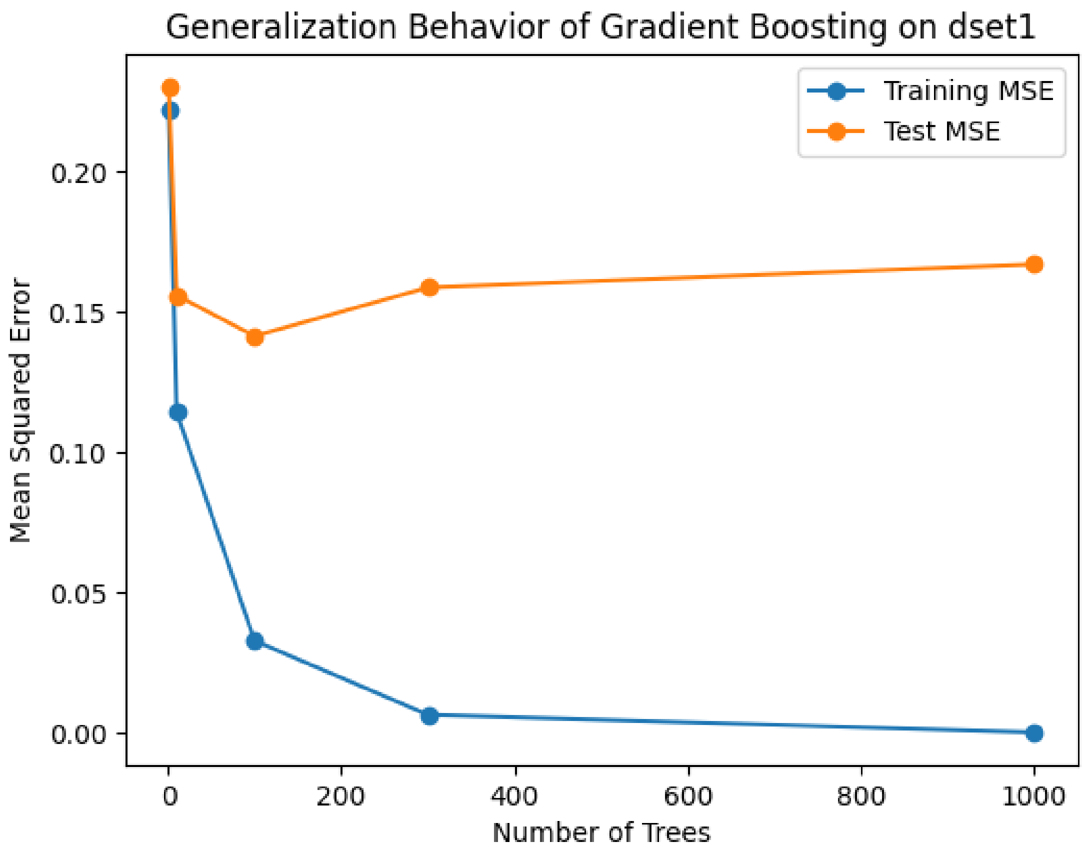
```
param_grid_rf = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10]
}

param_grid_knn = {
    'n_neighbors': [3, 5, 7, 9],
    'weights': ['uniform', 'distance'],
    'p': [1, 2]  # Manhattan distance (p=1), Euclidean distance (p=2)
}

# Initialize models
rf = RandomForestRegressor(random_state=42)
knn = KNeighborsRegressor()

# Perform grid search for each model
grid_search_rf = GridSearchCV(rf, param_grid_rf, cv=5,␣
 ↪scoring='neg_mean_squared_error', n_jobs=-1, verbose=1)
grid_search_knn = GridSearchCV(knn, param_grid_knn, cv=5,␣
 ↪scoring='neg_mean_squared_error', n_jobs=-1, verbose=1)

# Fit models on training data and use validation set for selecting the best␣
 ↪parameters
grid_search_rf.fit(X_train, y_train)
grid_search_knn.fit(X_train, y_train)

# Get best parameters and scores for each model
best_rf_model = grid_search_rf.best_estimator_
best_knn_model = grid_search_knn.best_estimator_

# Evaluate on the test set
rf_test_mse = mean_squared_error(y_test, best_rf_model.predict(X_test))
knn_test_mse = mean_squared_error(y_test, best_knn_model.predict(X_test))

# Results
results = {
    "RandomForestRegressor": {
        "Best Parameters": grid_search_rf.best_params_,
        "Validation Score (MSE)": -grid_search_rf.best_score_,
        "Test Score (MSE)": rf_test_mse
    },
    "KNeighborsRegressor": {
        "Best Parameters": grid_search_knn.best_params_,
        "Validation Score (MSE)": -grid_search_knn.best_score_,
        "Test Score (MSE)": knn_test_mse
    }
}

print(results)
```

```
Fitting 5 folds for each of 27 candidates, totalling 135 fits
Fitting 5 folds for each of 16 candidates, totalling 80 fits
{'RandomForestRegressor': {'Best Parameters': {'max_depth': None,
'min_samples_split': 2, 'n_estimators': 200}, 'Validation Score (MSE)':
np.float64(0.4197567709667808), 'Test Score (MSE)':
np.float64(0.3237004081632653)}, 'KNeighborsRegressor': {'Best Parameters':
{'n_neighbors': 9, 'p': 1, 'weights': 'distance'}, 'Validation Score (MSE)':
np.float64(0.5353664831969063), 'Test Score (MSE)':
np.float64(0.4464029344613551)}}
```

3 Generalization Behavior of Boosting

[ ]:
```
import pandas as pd
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# Load the datasets
dset0 = pd.read_csv('dset0.csv')
dset1 = pd.read_csv('dset1.csv')

# Separate the target variable (first column) from features (remaining columns)
X_dset0 = dset0.iloc[:, 1:]
y_dset0 = dset0.iloc[:, 0]
X_dset1 = dset1.iloc[:, 1:]
y_dset1 = dset1.iloc[:, 0]

# Number of trees to test
n_estimators_list = [1, 10, 100, 300, 1000]

# Function to fit Gradient Boosting model with different number of trees and␣
 ↪evaluate MSE
def evaluate_boosting(X, y, dataset_name):
    train_errors = []
    test_errors = []

    # Split data into training and testing sets
    from sklearn.model_selection import train_test_split
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)

    for n_estimators in n_estimators_list:
        # Initialize the model
        model = GradientBoostingRegressor(n_estimators=n_estimators,␣
 ↪random_state=42)
        model.fit(X_train, y_train)

        # Calculate MSE on both training and test sets
        train_mse = mean_squared_error(y_train, model.predict(X_train))
        test_mse = mean_squared_error(y_test, model.predict(X_test))

        train_errors.append(train_mse)
        test_errors.append(test_mse)

    # Plot the results
    plt.figure()
    plt.plot(n_estimators_list, train_errors, label='Training MSE', marker='o')
    plt.plot(n_estimators_list, test_errors, label='Test MSE', marker='o')
    plt.xlabel('Number of Trees')
    plt.ylabel('Mean Squared Error')
    plt.title(f'Generalization Behavior of Gradient Boosting on {dataset_name}')
    plt.legend()
    plt.show()

    return train_errors, test_errors

# Evaluate on dset0 and dset1
train_errors_dset0, test_errors_dset0 = evaluate_boosting(X_dset0, y_dset0,␣
 ↪"dset0")
train_errors_dset1, test_errors_dset1 = evaluate_boosting(X_dset1, y_dset1,␣
 ↪"dset1")
```


Generalization Behavior of Gradient Boosting on dset0
Training MSE
Test MSE
Mean Squared Error
Number of Trees
0.25
0.20
0.15
0.10
0.05
0.00
0
200
400
600
800
1000



3.1 Conclusion

- `dset0` **(Separable Data)**: This dataset likely has a clear underlying structure, allowing the model to generalize well with a moderate number of trees. It reaches a good margin without excessive complexity, and adding too many trees leads to overfitting.
- `dset1` **(Non-Separable or Complex Data)**: This dataset may have overlapping classes or complex patterns, causing the model to overfit quickly. The small margin between classes or clusters results in a test error that does not improve significantly even as the model complexity increases.

These results align with the theory that gradient boosting models can achieve better margins with more weak classifiers (trees), but this only holds if the data has a sufficiently large margin to exploit. In cases where the data is not separable, adding more trees mainly increases the risk of overfitting.

[ ]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader, random_split
import pandas as pd
import numpy as np
```

```python
# Load the dataset
data = pd.read_parquet("taxi_trips.parquet")
X = data.drop(columns="trip_duration").values
y = data["trip_duration"].values

# Convert data to tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32)

# Split dataset into training, validation, and test sets
dataset = torch.utils.data.TensorDataset(X_tensor, y_tensor)
train_size = int(0.7 * len(dataset))
val_size = int(0.15 * len(dataset))
test_size = len(dataset) - train_size - val_size
train_dataset, val_dataset, test_dataset = random_split(dataset, [train_size,␣
 ↪val_size, test_size])

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

# Define the neural network architecture
class UncertaintyModel(nn.Module):
    def __init__(self, input_dim):
        super(UncertaintyModel, self).__init__()
        self.hidden = nn.Linear(input_dim, 16)
        self.output_mean = nn.Linear(16, 1)
        self.output_log_std = nn.Linear(16, 1)  # log_std for stability

    def forward(self, x):
        x = torch.relu(self.hidden(x))
        mean = self.output_mean(x)
        log_std = self.output_log_std(x)
        std = torch.exp(log_std)  # Ensure positive std
        return mean, std

# Initialize the model, loss function, and optimizer
input_dim = X.shape[1]
model = UncertaintyModel(input_dim)
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Custom loss function based on Gaussian negative log-likelihood
def gaussian_nll_loss(mean, std, y):
    return torch.mean((y - mean) ** 2 / (2 * std ** 2) + torch.log(std) + 0.5 *␣
 ↪np.log(2 * np.pi))
```

```
# Training loop
def train(model, train_loader, val_loader, epochs=10):
    for epoch in range(epochs):
        model.train()
        for X_batch, y_batch in train_loader:
            mean, std = model(X_batch)
            loss = gaussian_nll_loss(mean.squeeze(), std.squeeze(), y_batch)
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        # Validation
        model.eval()
        val_loss = 0
        with torch.no_grad():
            for X_batch, y_batch in val_loader:
                mean, std = model(X_batch)
                val_loss += gaussian_nll_loss(mean.squeeze(), std.squeeze(),␣
 ↪y_batch).item()
        val_loss /= len(val_loader)
        print(f"Epoch {epoch+1}/{epochs}, Validation Loss: {val_loss:.4f}")

# Train the model
train(model, train_loader, val_loader)
```

```
Epoch 1/10, Validation Loss: 2.9607
Epoch 2/10, Validation Loss: 2.8664
Epoch 3/10, Validation Loss: 2.8592
Epoch 4/10, Validation Loss: 2.8382
Epoch 5/10, Validation Loss: 2.8510
Epoch 6/10, Validation Loss: 2.8392
Epoch 7/10, Validation Loss: 2.8241
Epoch 8/10, Validation Loss: 2.8219
Epoch 9/10, Validation Loss: 2.8191
Epoch 10/10, Validation Loss: 2.8081
```

```
[ ]: from scipy.stats import norm

     # Function to calculate probability of trip duration < 45 minutes
     def predict_probability(model, X):
         model.eval()
         with torch.no_grad():
             mean, std = model(torch.tensor(X, dtype=torch.float32))
         prob = norm.cdf((45 - mean.numpy().squeeze()) / std.numpy().squeeze())
         return prob

     # Evaluate on test set
```

```
test_probs = []
test_targets = []

for X_batch, y_batch in test_loader:
    probs = predict_probability(model, X_batch)
    test_probs.extend(probs)
    test_targets.extend(y_batch.numpy())

# Calculate accuracy: considering trips < 45 mins as '1' and >= 45 mins as '0'
threshold = 0.5
predictions = [1 if p >= threshold else 0 for p in test_probs]
actuals = [1 if y < 45 else 0 for y in test_targets]
accuracy = sum(int(pred == actual) for pred, actual in zip(predictions,␣
 ↪actuals)) / len(predictions)

print(f"Accuracy: {accuracy:.4f}")
```

```
C:\Users\a_i_b\AppData\Local\Temp\ipykernel_6732\1173317560.py:7: UserWarning:
To copy construct from a tensor, it is recommended to use
sourceTensor.clone().detach() or
sourceTensor.clone().detach().requires_grad_(True), rather than
torch.tensor(sourceTensor).
  mean, std = model(torch.tensor(X, dtype=torch.float32))
```

```
Accuracy: 0.9925
```