Predicting High Tides in Venice using Meteorological Data

Gisoo Shams
Bocconi University
May 2023

Introduction

The purpose of this project is to develop a machine learning model that predicts high tides in Venice using meteorological data. High tides in Venice can lead to flooding and other issues, so accurate predictions are crucial for planning and mitigating potential risks. My code implements a machine learning pipeline to train and evaluate a model for this task.

Data Preparation and Exploration

The code begins by loading the training and test datasets. Basic exploratory data analysis techniques are applied to gain insights into the data. The head, information, and descriptive statistics of the training dataset are displayed. A heatmap is used to visualize the correlation matrix and identify relationships between variables.

The first approach

Handling Missing Values:
Missing values in the training dataset are handled by filling them with the mean value of the respective column. This ensures that the data is complete and ready for further processing.

Data Preprocessing:
To ensure the numerical variables are on a similar scale, a MinMaxScaler is applied to normalize the training data. This step is crucial for improving the performance of certain machine learning algorithms that are sensitive to the scale of the input features.

Model Training and Evaluation:
A Random Forest classifier is instantiated and trained on the preprocessed training data. The trained model is then evaluated on a validation set. The accuracy score is computed to assess the model's performance in predicting high tides.

Hyperparameter Tuning:
Grid search is performed to find the best hyperparameters for the Random Forest classifier. This step helps optimize the model's performance by exploring different combinations of hyperparameters and selecting the ones that yield the best results.

```python
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.svm import SVC
from sklearn.metrics import classification_report

# Define the parameter grid for grid search or parameter distributions for random search
param_grid = {
    'C': [0.1, 1, 10, 20],
    'kernel': ['linear', 'rbf', 'poly'],
    'gamma': [0.001, 0.01, 0.1],
    'degree': [2, 3, 4],
    'class_weight': [{1: 5, 2: 10}]
}

# Create the SVM classifier
svm = SVC()

# Grid Search
grid_search = GridSearchCV(svm, param_grid, cv=5)
grid_search.fit(X_train_scaled, y_train)

# Grid Search
print("Best Parameters (Grid Search):", grid_search.best_params_)
print("Best Score (Grid Search):", grid_search.best_score_)

# Grid Search
best_model = grid_search.best_estimator_
y_pred = best_model.predict(X_test_scaled)
print("Classification Report (Grid Search):")
print(classification_report(y_test, y_pred))
```

```
Best Parameters (Grid Search): {'C': 1, 'class_weight': {1: 5, 2: 10}, 'degree': 2, 'gamma': 0.01, 'kernel': 'rbf'}
Best Score (Grid Search): 0.95
Classification Report (Grid Search):
              precision    recall  f1-score   support

           1       0.95      1.00      0.97       177
           2       1.00      0.57      0.72        23

    accuracy                           0.95       200
   macro avg       0.97      0.78      0.85       200
weighted avg       0.95      0.95      0.94       200
```

A picture of part of the hyperparameter tuning

Best Model Evaluation and Performance Metrics:
The best model obtained from the grid search is trained on the training data, and its performance is evaluated on the validation set. Precision, recall, and F1 score are calculated to provide a more comprehensive assessment of the model's predictive capabilities.

Cost Calculation:
Using a given cost matrix and the true and predicted labels of the validation set, the code calculates the overall cost. This cost metric provides a quantitative measure of the impact of misclassifications and aids in evaluating the practical effectiveness of the model.

Conclusion:
In conclusion, the provided code implements a machine learning pipeline for predicting high tides in Venice using meteorological data. It covers essential steps such as data loading, exploration, handling missing values, data preprocessing, model training and evaluation, hyperparameter tuning, and performance metric calculations.

The score of this approach was 0.14.

The second approach

Path Summary

Data Preprocessing:

- Reading the training data from a CSV file and split it into input features (x) and target variable (y).
- Handling missing values by filling them with zeros.
- Normalizing the input features using MinMaxScaler.
- Ratio 0.8 to 0.2 for train validation
- Target variable (y) -1 to further normalize the data.

Model Definition:
Here I defined a neural network model using PyTorch's nn.Module. The model consists of three fully connected layers with ReLU activation between them and a sigmoid activation at the output layer.
The neural network in this code is a feedforward neural network with three fully connected layers. It is defined in the **NeuralNetwork** class, which is a subclass of **nn.Module** from PyTorch. (The parent class for creating different neural networks)

The architecture of the neural network is as follows:

1. Input layer: The input layer has 37 nodes, which corresponds to the number of input features.
2. Hidden layers: There are two hidden layers with 64 and 32 nodes, respectively. Each hidden layer is followed by a ReLU activation function, which introduces non-linearity to the model.
3. Output layer: The output layer has 1 node, representing the predicted probability of a high tide. It uses a sigmoid activation function to squash the output between 0 and 1, providing a probability interpretation. For allocating the numbers to each class (0 or 1), we use the round() function that rounds to the nearest integer.

The forward propagation is implemented in the **forward** method of the **NeuralNetwork** class. It takes the input and passes it through the layers, applying the activation functions.
During training, the model uses the binary cross-entropy loss (**nn.BCELoss**) as the optimization objective. It aims to minimize the difference between the predicted probabilities and the target labels. The Adam optimizer (**optim.Adam**) is used to update the model's parameters based on the computed gradients. Also, the learning rate considered for this model is 0.001 which got computed and optimized with multiple models runs and observing the training loss reduction pattern.

To handle class imbalance in the target variable (high tides vs. low tides), the class weights are computed using the **compute_class_weight** function from scikit-learn. The weights are then used in the loss function to give higher importance to the minority class (high tides) for better training of the model. (Weights used in this function are from the weights=matrix)

The model is trained for a specified number of epochs (2500), with each iteration consisting of forward and backward propagation, loss calculation, gradient update, and parameter optimization. The training loop prints the loss value at regular intervals to further showcase our training progression.
After training the model, we use the validation dataset created at the start of our code to assess our model’s accuracy and report the results using simple accuracy (number of correct predictions/numbers of total predictions) and F1-score.
With multiple models runs and trying different tunings, we reach the suitable model tuning that can run our test dataset.

```
[11] # Create an instance of the neural network
     model = NeuralNetwork()

     # Define the loss function and optimizer
     criterion = nn.BCELoss(weight=sample_weights)
     optimizer = optim.Adam(model.parameters(), lr=10e-3)

     # Create a DataLoader for the training dataset
     # train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
```

```
# Train the model
num_epochs = 2500
for epoch in range(num_epochs):
    # outputs = torch.round(model(x_train_tensor))
    outputs = model(x_train_tensor)
    targets = y_train_tensor.view(-1)  # Convert targets to long data type and flatten
    loss = criterion(outputs.view(-1), targets)  # Reshape outputs to match target size
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    if epoch%100==0:
        print(f'In {epoch}|{num_epochs} the loss is: {loss.item()}')
```

```
In 0|2500 the loss is: 0.9225310683250427
In 100|2500 the loss is: 0.19541208446025848
In 200|2500 the loss is: 0.06606610119342804
In 300|2500 the loss is: 0.02567218989133835
In 400|2500 the loss is: 0.010800749063491821
In 500|2500 the loss is: 0.004987183026969433
In 600|2500 the loss is: 0.0024434856604784727
In 700|2500 the loss is: 0.0014024263946339488
In 800|2500 the loss is: 0.0009074033587239683
In 900|2500 the loss is: 0.0006332495249807835
In 1000|2500 the loss is: 0.0004643019055947661 4
In 1100|2500 the loss is: 0.0003577226889319718
In 1200|2500 the loss is: 0.0002832778263837099
In 1300|2500 the loss is: 0.0002307409449713304 6
In 1400|2500 the loss is: 0.0001913557789521291 9
In 1500|2500 the loss is: 0.0001609708124306053
In 1600|2500 the loss is: 0.0001380643370794132 4
In 1700|2500 the loss is: 0.0001173214404843747 6
In 1800|2500 the loss is: 0.0001018153561744839
In 1900|2500 the loss is: 8.873105252860114e-05
In 2000|2500 the loss is: 7.80350310378708e-05
In 2100|2500 the loss is: 6.931291136424989e-05
In 2200|2500 the loss is: 6.141065387055278e-05
In 2300|2500 the loss is: 5.4910677135922015e-05
In 2400|2500 the loss is: 4.934856042382307 4e-05
```

- A picture of part of the model training

Validation:

- Normalizing the validation data and evaluating the trained model's performance on the validation set.
- Calculating the validation loss, accuracy, and F1 score.

```
# Evaluate the model on the validation set
model.eval()
total_val_loss = 0
correct = 0
total = 0
val_criterion = nn.BCELoss()
with torch.no_grad():
    for inputs, targets in val_loader:
        outputs = model(inputs)
        predicted = torch.round(outputs).squeeze().long()  # Squeeze and convert to long
        total_val_loss += val_criterion(outputs.view(-1), targets).item()
        total += targets.size(0)
        correct += (predicted == targets).sum().item()

val_loss = total_val_loss / len(val_loader)
accuracy = correct / total

print(f'Validation Loss: {val_loss:.4f}')
print(f'Correct Predictions: {correct}/{total}')
print(f'Validation Accuracy: {accuracy * 100:.2f}%')
```

```
Validation Loss: 0.5132
Correct Predictions: 189/200
Validation Accuracy: 94.50%
```

```
[15] f1_score(np.array(y_val_normalized).reshape(-1,1), torch.round(model(x_val_tensor)).detach().numpy(), average='weighted')

     0.9444663536776212
```

```
[16] x_test = pd.read_csv('test.csv')
     x_test = x_test.fillna(0)

     x_test_normalized = scaler.fit_transform(x_test)

     # Convert x_train and y_train to PyTorch tensors
     x_test_tensor = torch.Tensor(x_test_normalized)
     x_test_tensor

     tensor([[0.7671, 0.3034, 0.3297,  ..., 0.4099, 0.0000, 0.0000],
             [0.3591, 0.6307, 0.6129,  ..., 0.1156, 0.0000, 0.0000],
             [0.4859, 0.2978, 0.2459,  ..., 0.0663, 0.0000, 0.0000],
             ...,
             [0.9036, 0.5929, 0.3801,  ..., 0.1531, 0.0000, 0.0000],
             [0.3665, 0.5872, 0.4438,  ..., 0.0816, 0.0000, 0.0000],
             [0.8895, 0.3518, 0.2517,  ..., 0.0221, 0.0000, 0.0000]])
```

```
[17] test_out = torch.round(model(x_test_tensor)).detach().numpy()
     test_out = test_out + 1
```

A picture of part of evaluating the model

Prediction:

- Reading the test data from a CSV file, preprocessing it, and making predictions using the trained model.
- Converting the predicted values to the original target variable format and save the predictions in a text file.

The score of this approach was 0.12.

Methods Description

- torch.Tensor: Converts a numpy array or Python list to a PyTorch tensor.
- torch.nn.Module: A base class for all neural network modules in PyTorch.
- nn.Linear: Applies a linear transformation to the input data.
- nn.ReLU: Applies the rectified linear unit activation function element-wise.
- nn.Sigmoid: Applies the sigmoid activation function element-wise.
- torch.optim.Adam: Implements the Adam optimizer for stochastic optimization.
- torch.utils.data.DataLoader: Combines a dataset and a sampler, providing an iterable over the given dataset.
- nn.BCELoss: Computes the binary cross-entropy loss between input predictions and target labels.
- MinMaxScaler: Transforms features by scaling each feature to a specified range.
- train_test_split: Splits arrays or matrices into random train and validation subsets.
- compute_class_weight: Computes class weights for imbalanced datasets based on the target