Table of contents

Final Report

Author

Olivia Armstrong & Connor Martindale

Introduction

Extreme weather events can have disastrous effects on numerous aspects of our planet. From tropical storms to wildfires, these events are sweeping across the globe in an alarmingly frequent fashion. With this inevitable threat, it becomes increasingly important to examine these trends so that proper preparation can be taken. While there are current models that exist to help predict weather conditions, these models have a limited forecast horizon. With the increased availability of meteorological data, we can utilize machine learning techniques to help improve these weather forecasts. These longer weather forecasts can help effectively prepare communities for various weather conditions.

The dataset provided for this competition was created in collaboration with Climate Change AI (CCAI). Provided was pre-prepared training data, testing data, and a sample solution. Each row in the data corresponds to a single location and a single start date for a two-week period. Additionally, weather and climate related information was included in these rows. The task at hand was to predict the arithmetic mean of the maximum and minimum temperature over the next 14 days, for each location and start date.

Data: Overview

The dataset was given as a separate training and testing set. The training set contains 103,659 entries with 246 features per entry. The testing set has 31,354 entries with 245 features as the values of the target variable that we are predicting are not included in the testing set.

Within the dataset are weather readings for 575 different locations within the United States. The features in the dataset represent average readings taken from weather instruments in these unique locations over a two week period. Each location in the training set had data from multiple different periods spanning from September 1, 2014 to August 31, 2016. The testing set contains data from November 1, 2022 to December 31, 2022.

Our target variable is [contest-tmp2m-14d__tmp2m], and it represents the mean value of the maximum and minimum values for the observed temperatures during each two week period for each location. Therefore, we are using various readings from weather-related instruments in order to predict the average temperature in an area during a two week period.

The key variables for understanding the dataset are [lat], [lon], [startdate], [climateregion__climateregion], and the target variable. Location data is given by the [lat] and [lon] features (the latitude and longitude of the area). The [startdate] feature represents the first day of the two week period. Lastly, [climateregion__climateregion] contains a 3 letter label for 14 different climate regions. The remaining variables in the dataset are continuous, numerical features from the weather instruments.

Data: Cleaning

After checking the data types of all of the features, we noted that our first step would be to convert all variables to a usable data type.

First, we converted the [startdate] feature to 3 separate variables ([day], [month], and [year]) to allow the model to distinguish more clearly between the date information.

From our exploration of the datasets, we learned that the testing set only covers the months of November and December, while the training set covers two full years. Therefore, we determined that we should create an additional variable representing the season using the entry's month so that we could train the model using only the data from the same season (winter) as in the testing set.

Next, we used the latitude and longitude variables to create a new variable that would provide a numerical label to each unique location within the dataset.

We then converted the [season] and [year] variables to dummy variables of a numerical format and changed them to be boolean variables.

Having finished creating additional variables and changing data types, we then dropped all of the rows with missing data values within the data set.

Lastly, we Z-scored all of the numerical data to provide the model with a more standardized version of the data.

Modeling: Baseline Lasso Model

In order to provide a baseline assessment to compare against our more complex model, we created a Lasso regression model. We chose to use Lasso regression because of its ability to eliminate the features in the dataset that it deems are least important. By doing so, Lasso provides us with some limited regularization for our model without us having to manually tune the model.

Our first attempt with the Lasso model provided a training set RMSE score of 2.513 (with RMSE as our evaluation method, lower numbers represent improved performance). However, our testing set RMSE was 12.986, which showed us that the model was extremely overfit to the training set as the training predictions were far more accurate than the testing predictions.

Next, we removed all of the date-related information from the dataset and ran the Lasso model again. The overall performance of the model decreased, and the overfitting problem remained, however.

Our next step was to run the model without any location-based data. At this stage in our process, we were attempting to discover the source of our overfitting problem by removing certain variables and rerunning the model. After removing the labels for each unique location (a variable that we created), the model was no longer severely overfit.

Having eliminated our overfitting problem, we could then test our aforementioned hypothesis: since the testing data only contains data from the months of November and December, the model might perform better when it has only been trained on data from Autumn and Winter months (or only on data from November and December).

First, we filtered the dataset so that only the data taken during Autumn and Winter remained. Unfortunately, the model became extremely overfit. When testing with only data from November and December, the model was still overfit to the training set as the training RMSE was 3.149 and the testing RMSE was 7.008, showing that the model was performing significantly better on the training set.

Therefore, the conclusion that we learned from our baseline Lasso model was that we should revert some of the data cleaning and manipulation in order to prevent overfitting.

The final version of the dataset that we used merely had dummy variables created for the climate regions, and the [startdate] variable was split into three separate variables (day/month/year). Missing values in the dataset were dropped.

When testing our Lasso model with this final iteration of the dataset, we received the best results so far: our training RMSE was 1.160 and our testing RMSE was 1.611.

```
#Load the train and test sets
train = pd.read_csv('train_data.csv')
test = pd.read_csv('test_data.csv')
train = pd.get_dummies(train, columns = ['climateregions__climateregion'], drop_first = True)
test = pd.get_dummies(test, columns = ['climateregions__climateregion'], drop_first = True)
#Convert startdate variable to a useable type: new columns for day, month, year
train.startdate = pd.to_datetime(train.startdate)
train['day'] = train.startdate.dt.day
train['month'] = train.startdate.dt.month
train['year'] = train.startdate.dt.year
train = train.drop(['startdate'], axis = 1)

test.startdate = pd.to_datetime(test.startdate)
test['day'] = test.startdate.dt.day
test['month'] = test.startdate.dt.month
test['year'] = test.startdate.dt.year
test = test.drop(['startdate'], axis = 1)
#get rid of null values
train.dropna(inplace = True)

#split up training predictors and target values
X_train = train.loc[:, train.columns != 'contest-tmp2m-14d__tmp2m']
y_train = train['contest-tmp2m-14d__tmp2m']

#testing data rename
X_test = test

#exclude object types since regression can't handle them (if any remain)
X_train = X_train.select_dtypes(exclude=['object'])
X_test = X_test.select_dtypes(exclude=['object'])
lasso6 = Lasso()
lasso6.fit(X_train, y_train)

print("TRAIN RMSE: ", mean_squared_error(y_train, lasso6.predict(X_train), squared = False))
```

With our baseline model finalized, we could now move on to creating more complex models in an attempt to surpass the Lasso regression model's performance.

Modeling: Random Forest

```
To begin, we chose to use a Random Forest model to compare against our Lasso regression. Random Forest models utilize multiple variations of a single model in order to create an average that is used to make predictions. Therefore, this type of model is well-suited for reducing the tendency for machine learning models to overfit to the training set.

randForest = RandomForestRegressor(max_depth = 5, random_state = 0)
randForest.fit(X_train, y_train)

Our model created 100 different Decision Trees, each with a maximum depth of 5 levels (each tree would make up to 5 decisions). The training RMSE of the Random Forest was 2.410, and the testing RMSE was 2.459, showing that, while the model was not overfit, the model failed to outperform our Lasso Regression.
```

Modeling: PCA and Lasso Regression

```
Since it appeared that our Lasso Regression model was more accurate than we had realized, we decided to use Principal Components Analysis in an attempt to improve our Lasso model. The benefit of using PCA is that this technique will combine like variables and their influences on the model into Principal Components, thus lowering the total number of features used in a model. This technique is useful for combatting overfitting and distilling the important information found in a large data set.

Our PCA created 41 principal components that could explain 90% of the variance in the data, effectively lowering the number of features in the dataset from 260 to 41. When we used this altered version of the data set with a Lasso regression model, the model had a training RMSE of 2.598 and a testing RMSE of 2.426. Using PCA in conjunciton with Lasso regression ultimatley lowered the effectiveness of the model as less data was being used to create the predicitons.

Since both PCA and Lasso regression are used to reduce overfitting (as they are both regularization techniques) at the potential cost of lowering the overall accuracy of the model, we now realize that it may have been unwise to use both techniques together. Therefore, in a future test, we will use a different model in tandem with the Principal Components Analysis.
```

Modeling: XGBoost Model

```
 For our modeling, we decided to utilize XGBoost. XGBoost has various benefits that make it ideal for the data science problem at hand. One of which includes that it is able to handle large datasets efficiently as it can automatically perform parallel computation, meaning many calculations are carried out at the same time.

Additionally, the numerous hyperparameter options allow the opportunity to further improve model performance through the process of tuning. Furthermore, XGBoost implements the gradient boosting algorithm. To summarize, this algorithm works by adding new models to correct errors made in pre-existing models. The final prediction is the average of all the models predictions, making it ideal for regression problems like the one at hand.


When researching how to implement this model, the high number of customizable hyperparameters made it evident that a tuning technique would be necessary to find the most optimal values. At first, we explored simulated annealing. Simulated annealing finds the most optimal hyperparameters in a model by finding the global optima of a function, simulating thermodynamics. Simulated annealing has the benefit of being very effective in the world of machine learning. However, after exploring this option we quickly realized it would not be an ideal choice as there is no package for it and instead the algorithm would have to be implemented by hand. Other options for hyperparameter optimization include GridSearch and RandomizedSearchCV.


Ultimately, we ended up choosing to utilize RandomizedSearchCV. This method works by randomly passing the set of hyperparameters, calculating the score, and then outputting the set of hyperparameters that have the best score. RandomizedSearchCV has several parameters including the parameter distributions and number of iterations. The parameter distribution is the dictionary of parameters that are being optimized. For our XGBoost model, we choose to optimize the max_depth, subsample, colsample_bytree, learning_rate, gamma, and scale_pos_weight. As overfitting was a big concern for our model, optimizing these important parameters could help mitigate this.

# Parameter Dictionary
param_dict = OrderedDict()
param_dict['max_depth'] = [5, 10, 15, 20, 25]
param_dict['subsample'] = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
param_dict['colsample_bytree'] = [0.5, 0.6, 0.7, 0.8, 0.9, 1.0]
param_dict['learning_rate'] = [0.01, 0.05, 0.10, 0.20, 0.30, 0.40]
param_dict['gamma'] = [0.00, 0.05, 0.10, 0.15, 0.20]
param_dict['scale_pos_weight'] = [30, 40, 50, 300, 400, 500, 600, 700]
# Model Building
rs_model=RandomizedSearchCV(classifier,param_distributions=param_dict,n_iter=5,scoring='roc_auc',n_jobs=-1,cv=5,verbose=3)
# Fit
rs_model.fit(X_train,y_train)
rs_model.best_estimator_


When we optimized our parameters and printed the results, we could see that the best values were colsample_bytree = 0.8, gamma = 0.1, max_depth=25, scale_pos_weight = 50, and subsample = 0.7. Following this step, we were able to create our XGBoost Model. Using these parameter values for our XGBoost model demonstrated that the model performed very well on our training set, with a RMSE of 0.868332. However, as this value is so low, this did cause concern for how it could perform on the testing set. Our concerns regarding this model's tendency to overfit were justified as the model's testing RMSE was 2.294, which was far greater than the trainign RMSE of 0.868.

# Create Model w/ Optimized Parameters

reg = xgb.XGBRegressor(base_score=0.5, booster='gbtree',
                       n_estimators=1000,
                       early_stopping_rounds=50,
                       objective='reg:linear',
                       colsample_bytree=0.8,
                       gamma = 0.1,
                       max_depth=25,
                       scale_pos_weight=50,
                       min_child_weight = 1,
                       subsample=0.7,
                       learning_rate=0.01)

reg.fit(X_train, y_train,
        eval_set=[(X_train, y_train), (X_train, y_train)],
        verbose=100)
```

Improvements

```
There are numerous ways we could improve our XGBoost model given the high amount of parameters that could be customized. To experiment with them all would possibly yield better results. Additionally, while RandomizedSearchCV is a great parameter optimization method as it is easy to implement, there are other methods that could prove to be better.

Grid Search is one of those methods as it performs a search for every single combination of parameter values, making it highly accurate. However, this benefit comes with some major drawbacks, including the fact that it is computationally expensive. For a model that has so many parameters and associated values, this was a concern.
```

Limitations

```
One of the limitations that was encountered during this was the fact that the dataset included a large number of variables which made it difficult to determine what was relevant in what we wanted to predict.


Additionally, time proved to be a major limitation. XGBoost proved to be difficult to experiment with as the large dataset made it slow to run, which in turn limited our ability to tune it and experiment with it more. While RandomizedSearchCV is a great method for hyperparameter tuning, it is based on luck meaning it may have not been the most effective way to optimize these parameters utilized in the model. To combat these limitations, we would use a subsection of the dataset when tuning our model as opposed to using the full dataset for each iteration.


The reason we chose XGBoost was because we wanted to gain experience with a model/algorithm that we had never used before. While this was a good learning experience, this in itself did prove to be a limitation as we did not have much time to explore the most effective ways to implement it. If given more time, we may have been able to create a better model.
```

Kaggle Submission

```
With a testing RMSE of 1.611, our Lasso model put us in 148th place out of 183 submissions in the Kaggle competition. Our XGBoost model had a testing RMSE of 2.294, and since this score is worse than the Lasso model's RMSE, the Lasso model ended up being our final submission.
```

Conclusion

```
When presented with this challenge, we took what we thought was the most unique approach to the issue. XGBoost was an unfamiliar model for us both. By pursuing this route, we thought this may be interesting to others who took different approaches. This in itself opens up different paths of thinking when approaching a machine learning problem.
```