660-Final-Project

Project Type: (1) Design a system based on real-world data
Number of Student Authors: 2

Armin Bazarjani (bazarjan@usc.edu)
Cameron Akhavan (cakhavan@usc.edu)

January 11, 2021

1 Abstract

The goal of this project was to be able to train a classifier to predict whether there will be a forest fire given a place's city location, date, time, and weather information such as temperature, humidity, pressure, etc.

In order to gather the most fruitful data to train our classifier, we decided to combine datasets that specialized in either forest fire data or weather data. Given the complexity and diversity of the datasets, preprocessing and merging the data into one dataset proved to be a very complicated task.

Given the complexity of the resulting dataset, we experimented with classification models with varying types of decision boundaries: Logistic Regression with l1 and l2 regularization for a linear boundary, Random Forest and Adaboost for parallel-axis boundaries, and svm with gaussian kernel for a nonlinear boundary.

Derived from the master dataset, we made sure to train each model and determine hyperparameters using 5 Fold Cross Validation using a training set, select the best model from a validation set, and score the performance of the final model with a test set.

In the end, we found that our Random Forest model performed the best with a better than expected performance of 74% accuracy in classifying whether a fire will begin on our test set.

2 Introduction

2.1 Problem Type, Statement and Goals

As global warming continues to increase temperatures around the globe, natural disasters such as forest fires have increased at catastrophic rates. Since the dawn of the industrial revolution, the burning of fossil fuels and subsequent

emission of $C0_2$ has increased at an average rate of 0.07°C (0.13°F) per decade since 1880; however, the average rate of increase since 1981 (0.18°C / 0.32°F) is more than twice as great [Lindsey, Dahlman 2020]. As we've seen in recent years, forest fires have ravished dry and hot climates such as California to the point where a code red emergency forest fire has just as much surprise on its residents as the morning sun. We are far from solving the root cause of our problem, but there are measures we can take to combat the ever growing damage that we must currently face.

Currently, the USDA Forest Service's Wildland Fire Assessment System compiles weather data from over 1500 weather data sensors to compile a topological map assessing forest fire conditions[WFAS 2020]. As weather data is very inexpensive and accessible, many researchers rely heavily on it to make their forest fire forecasts; however, researchers still struggle everyday to reliably and accurately predict forest fires and is often common practice to make their weather data publicly available to open source their efforts to the public. The goal of this project is to provide preventative and proactive measures in fighting forest fires by predicting when and where they will occur before they happen. By compiling data sets from various sources specializing in either forest fire data collection or weather data collection, we hope to create a new dataset that provides exceptional information for various statistical classification techniques.

This project will prove very challenging for a multitude of reasons. The most apparent obstacle is the nature of predicting something as complex as a forest fire with weather data which is even more complex in itself. It will prove very interesting being able to generate a reasonably accurate predictor using very cheap data such as weather instead of relying on expensive satellite sensors and topographic maps containing vegetation specifics such as forest floor dry brush density. This feat will require careful attention to maximize the predictive performance of our models as well as our model selection decisions.

Another major challenge of this project will be combining datasets from different sources with different structures in the most optimal way. We aim to combine two datasets that are both large and specific to one domain (either forest fire data or weather data), while sharing enough geographical and temporal data to be able to compare them.

After much research, we decided to pull our forest fire data from The Fire Program Analysis fire-occurrence database (FPA FOD) which contains over 1.88 million geo-referenced wildfire records from the years 1992 to 2015.

For our weather data we are going to use The Kaggle Historical Hourly Weather Data 2012-2017 contains high temporal resolution (hourly measurements) data of various weather attributes.

Combining these two datasets will require a significant amount of preprocessing. We will need to carefully analyze the structure of each dataset and accordingly restructure both to cohesively merge into a singular dataset that maintains their information and feature relationships. This is all while finding an intersection of data points between all datasets that preserves the most amount of data points for fruitful analysis.

2.2 Previous Work - None similar to our approach

2.3 Overview of Our Approach

The initial step to our approach involved modifying the structure of our weather datasets and fire dataset to merge cohesively into a single dataset. This required carefully sorting column names and feature values from both and pruning any information not common to both (non-overlapping years, locations, etc).

The most crucial part of restructuring the two datasets was making sure that the forest fire dataset was appended to the weather data in a way that would characterize each sample of time, location, and weather data by either having a presence of fire or not.

We expected our data to have a strong correlation with not only weather and geographical features, but temporal features as well. Temporal features such as hours, days, and months are cyclical in nature so we made sure to preserve this in our encoding.

As our dataset became increasingly complex and abstract from merging, we tried many models with various decision boundaries. To separate the fire class from non-fire class, we used logistic regression for a linear decision boundary, adaboost and random forest for axis-parallel boundaries, and svm with gaussian kernels for non-linear decision boundaries.

Also, for each model we used 5 Fold Cross Validation techniques on a Training set for determining hyperparameters, validation set for model selection, and a test set for final model assessment.

3 Implementation

3.1 Data Set

As mentioned before, we initially started with the Kaggle Historical Hourly Weather Data 2012-2017 for our weather data and the FPA FOD dataset for our fire data.

The Kaggle Historical Hourly Weather Data 2012-2017 contains high temporal resolution (hourly measurements) data of various weather attributes, such as temperature, humidity, air pressure, etc for 30 US cities. This dataset consists of 7 CSV files compiled from various sources containing only its respective weather feature reading(temperature, humidity, etc). Each CSV file had one column DATETIME and the rest of the columns labels were city names. We combined and restructured all CSV files into one dataset that resembles the format of the feature column in the chart below.

The Fire Program Analysis fire-occurrence database (FPA FOD) contains over 1.88 million geo-referenced wildfire records, representing a total of 140 million acres burned during the 24-year period (1992 to 2015). This is an SQL database with an inherently unique dataset structure. While this data contains

over 50 features, we decided to only use fire occurrence, fire location, and discovery date for our analysis. We chose this not only to reduce the complexity of our dataset, but decided that features such as fire size and fire reporting agency were not important for our focus.

The final combined master dataset was the product of merging these two datasets by finding all dates and times of forest fire occurrence from FPA FOD, matching them with samples from our weather dataset that had equivalent datetime values as well as geographical location, and appending a new class column of fire (occurrence) being true or false.

Table 1: Feature Table

Feature	Data Type	Cardinality/Range	Min/Max	Description
humidity	real (double)	0.00-100.0	(5.0-100.0)	% of absolute humidity
pressure	real (double)	R	(886.0-1059.0)	Air Pressure (hPa)
temperature	real (double)	R	(250.774-314.4)	Kelvin (K)
weather	Categorical (string)	35	N/A	Description of Weather
wind direction	real (double)	(0.0-360.0)	(0.0-360.0)	meteorological degrees (°)
wind speed	real (double)	R	(0.0-50.0)	meters per second (m/s)
city	Categorical (string)	8	N/A	Nearest city of Occurrence
HOUR	real (int)	0-23	0-23	Hour of Occurrence
DAY	real (int)	1-31	1-31	Day of Occurrence
MONTH	real (int)	1-12	1-12	Month of Occurrence
YEAR	real (int)	2012 - 2015	2012 - 2015	Year of Occurrence
FIRE	Categorical (string)	2	N/A	Forest Fire Occurrence

3.2 Data set Methodology

After finding all data points common to our original datasets, our master dataset had 227,800 data points.

We split our master dataset into a Training test set of blank 145,792 (67%), a validation set of 36,448 data points, and a test set with 45,560 data points(33%).

While looking at our Training set, we noticed we had significant class imbalance (1,167 positive cases and 144,625 negative cases).

To accommodate this issue we experimented with different resampling methods. We created two new training sets: an oversampled training set using SMOTE to bring the minority class up to the majority class (resulting in 144,625 samples each) and an undersampled training set using random datapoint deletion from the majority class (resulting in 1,167 samples each).

For each model we trained, we did a GridSearch stratified 5 Fold Cross Validation with both the undersampled and oversampled training data (separately). For each dataset, this entails giving a GridSearch algorithm a list of hyperparameters for each model. For each configuration of hyperparameters from the list, the given training set was then split into 5 randomly shuffled stratified folds. For each fold, one of the folds was used as a validation set (for training) while the other 4 folds were used for training the model. The resulting training and training validation scores for each model were then used to determine which hyperparameters were best for the model.

Algorithm 1 GridSearchCV for Finding Best Parameters

function CVPARAMS(X_train,y_train,$input\_grid$,$model$) ▷ $input\_grid$ are the different parameters to test
    $kfold = StratifiedKFold()$
    $grid\_cv = GridSearchCV(kfold, input\_grid, model)$
    $grid\_cv.fit(X\_train, y\_train)$
    $best\_params = grid\_cv.best\_params\_$
    return $best\_params$
end function

This procedure gave us the best hyperparameters for each model for each training dataset (oversampled or undersampled). Lastly, each finalized model was then scored on the validation set to compare our final results.

Unfortunately, we were only able to include our results from the undersampled data as running our models on the oversampled dataset for over 75 hours was not enough time to finish running all of our models. Thus, below are figures illustrating the procedures with and without oversampled data.

As we only are including the undersampled training set and had 5 finalized models for the set, the validation set was used 5 times. The model that performed the best on the validation set was selected as our final model. We then used the Test set only once to score the performance of our final model.

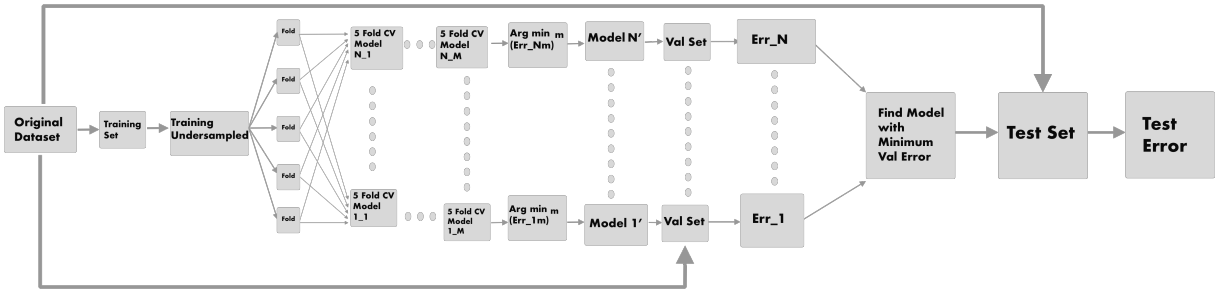

Figure 1: Undersampling Methodology with N models and M combinations of HyperParameters

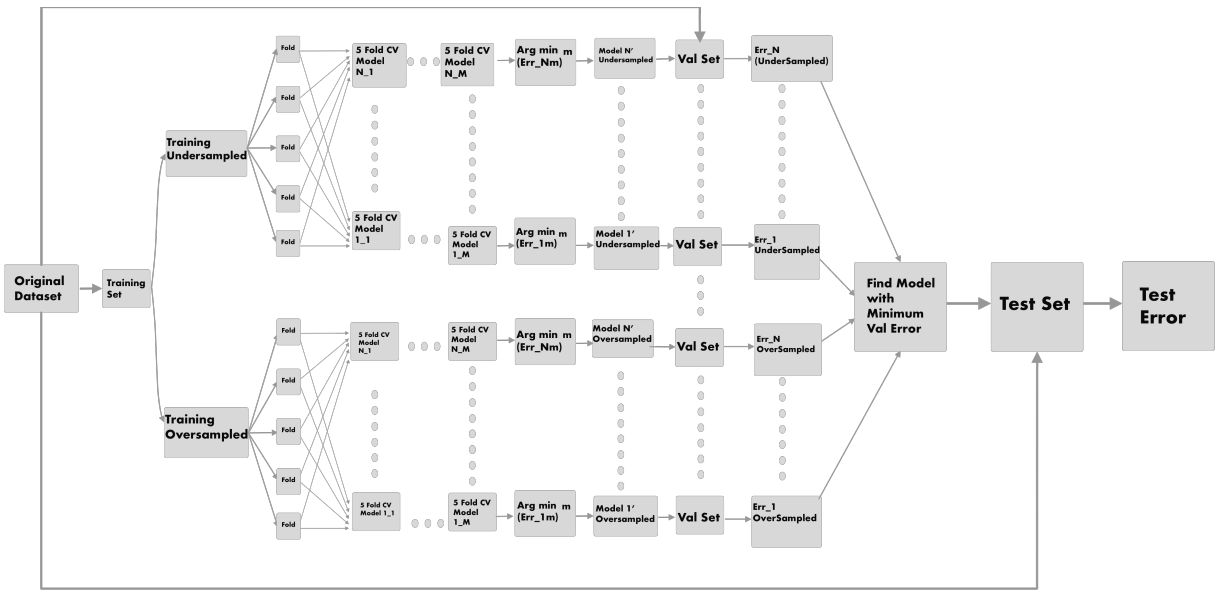

Figure 2: Oversampling and Undersampling Methodology with N models and M combinations of HyperParameters

3.3 Preprocessing, Feature Extraction

We decided it would be best to preprocess the data after we combined all the different data sets together into a sort of "master" data set. The preprocessing was relatively straightforward.

After further inspection of the master dataset, we noticed that most of our NaN values occurred where a single datapoint was missing it's entire feature space. Instead of imputing this, we decided it would be best to filter these special cases out and completely remove them instead. The rational for this is that these datapoints contained nothing more than a date and a city. We thought that imputing at this point would give the same results as an oversampling method like SMOTE and wouldn't be too useful at this stage of the project.

Luckily, this wasn't a prevalent condition and it didn't make us lose too much data. Afterwards, we were still left with some NaN values that we settled on imputing in a more traditional way. To do this, we used the MICE (Multiple Imputations by Chained Equations) implemented in sklearn's iterative imputer class (see *algorithm* 1 for more details). The way the iterative imputer works, is that it models each feature with missing values as a function of the other features in a round robin fashion.

Algorithm 2 Multivariate Imputation using MICE

```
function MvImp(df_incomplete)   ▷ Where df_incomplete is a dataFrame containing NaN values
    df_incomplete.fillna(9999)
    imputer = IterativeImputer()
    imputer.fit(df_incomplete)
    imputed_data = imputer.transform(df_incomplete)
    return imputed_data
end function
```

After imputing the missing data we moved onto feature extraction. The only feature extraction we performed was to utilize our DATETIME object. Instead of just using it as an index to combine the different datasets, we decided it might be beneficial to encode the date in a way that preserves its cyclical significance. This allows us to take advantage of potential hidden patterns in the dataset. We thought this would be especially important for predicting fires at is well known that there are certain times of the year nicknamed the "Fire Season" where fires become more prevalent.

In order to do this, we first extract the hour, day and month from the DATETIME object, normalize them to be between 0 and 2π. We then used the *sin* and *cos* functions on each of them to create six additional features.

Afterwards, we standardized all real-valued features using sklearn's StandardScaler class. For the categorical data, we decided to use value encoding, again assisted by sklearn, however this time from the LabelEncoder class.

For calculating feature importance, we used three different methods, all of which were assisted through use of sklearn classes. The first method we used

was to fit 5 RandomForest Classifiers to our dataset. Afterwards we outputted the average importance of each feature as was calculated from the classifiers. The second method was to do the same using L1 weighted Logistic Regression. Afterwards, we observed which values tended towards zero more often than not. The third method of calculating feature importance was to create a Pearson correlation heatmap.

After all three feature importance methods were completed, we used our judgement to decide which features were worth keeping and which we were better off without.

3.4 Training Process

Before discussing the training process, a necessary disclaimer must be made. All of the parameters that were chosen for the models were arrived at through our validation process that is documented at the end of this section (algorithm 2). Also, we will reference the results throughout this section, however we thought it would be best to refrain from showing the classifier results until the next section where we could go into more detail about them.

Logistic Regression

The first model we used was a logistic regression model with both L_1 and L_2 weighted penalties. The choice for this model was because we thought we could use it as our general baseline model to compare the classification results with those of other models.

We used the Limited-memory Broyden–Fletcher–Goldfarb–Shanno (lbfgs) solver for the L_2 model and the SAGA solver for the L_1 model. Both solvers were implemented with a cross-entropy loss. The lbfgs solver is essentially the same as using Newton's Method, the only difference being that the Hessian matrix is *approximated* using updates that were specified by gradient evaluations. Or, in other words, by using an estimation to the inverse of the Hessian matrix. The SAGA solver is a variant of the stochastic average gradient (SAG) solver, essentially a variation of gradient descent that uses a random sample of previous gradient values.

Through the cross-validation function for finding the best parameters (algorithm 2), we found the best value for λ_1 to be 0.01 for our L_1 penalized model and the best value for λ_2 to be 10 for our L_2 penalized model. sklearn allows for further parameter modification, however we found the other parameters to have little effect on the results.

As far as overfitting was concerned, we weren't. The λ_1 and λ_2 regularizers would help guard against overfitting to the extent that by trying out a wide range of them, our cross-validation procedure would determine the one that protected against overfitting the most and simultaneously gave us the best results.

Mathematically, the cost function of logistic regression becomes:

$$J(\theta) = \frac{1}{m}[\sum_{i=1}^{m} -y_i log(h_\theta(x_i) + (1-y_i))log(1-h_\theta(x_i))] + \frac{\lambda}{m}\sum_{j=1}^{n}|\theta_j|$$

for L_1 penalized logistic regression and,

$$J(\theta) = \frac{1}{m}[\sum_{i=1}^{m} -y_i log(h_\theta(x_i) + (1-y_i))log(1-h_\theta(x_i))] + \frac{\lambda}{2m}\sum_{j=1}^{n}\theta_j^2$$

for L_2 penalized logistic regression.

Random Forest

The second model we used was a Random Forest model. Our choice to use a random forest model was for the benefits of being robust to outliers, not being as susceptible to overfitting as other models, and being able to extrapolate to predictions for data that is outside of the bounds of our training data.

As a random forest model is an estimator that fits multiple decision tree classifiers on various sub-samples of the data. There are many parameters that we can alter. After looking online sklearn's documentation, we learned that although there are many parameters, the main ones that directly make an impact on the model are the number of trees in the forest ($n\_estimators$), the maximum depth of the tree (max_depth), and the maximum number of features to consider when looking for the best split ($max\_features$). Through the cross-validation function we found the best parameters to be $max\_depth = 60$,n_estimators=500, and $max\_features = sqrt(max\_features)$.

Again, with a Random Forest we were not entirely concerned about overfitting as when you add more trees to a model you are less likely to overfit. We then used a similar approach as in logistic regression, casting a wide net of trees to use in our parameter search range. This will again allow our cross-validation procedure to determine for us the best number of trees that will do well on the validation data which would simultaneously imply that it is most likely not overfitting.

We thought that the most enlightening math would be to show the final decision rule of a random forest:

$$C_{rf}^{B}(x) = majorityvote[C_b(x)]_1^B$$

where $C_b(x)$ is the class prediction of the bth random forest on input x, and B is the number of trees.

AdaBoost

The third model we used as an AdaBoost model. Our thought process with choosing the AdaBoost classifier is that after seeing the promising results of our Random Forest classifier, we thought that the natural progression would be to test out the AdaBoost classifier next as we were convinced at this point of the efficacy of using decision trees. And of course, this was implemented using sklearn's AdaBoost classifier class.

Adaboost, for a quick introduction, is a classifier that can be considered a "meta-estimator" that first fits a single classifier to the dataset. It then fits additional copies of the classifier to the dataset, except now the weights are

adjusted by the incorrectly labeled instances. This is done in the hopes of generating more focus on the difficult cases to make the overall decision more accurate.

With AdaBoost there weren't too many parameters to change. Sklearn allows you to change the *base_estimator* that the model us built on top of. However, as previously stated, my partner and I were happy with the way decision trees were performing, so we left that to the base case which was, $DecisionTreeClassifier(max\_depth = 1)$. As for the other parameters, we again consulted sklearn's documentation and found that the most influential parameters were the maximum number of estimators at which boosting is terminated ($n\_estimators$), and the learning rate which shrinks the contribution of each classifier ($learning\_rate$). Through our cross-validation function we found the best parameters to be $n\_estimators = 200$ and $learning\_rate = 0.1$.

In similar fashion as with random forests, the final decision rule of the AdaBoost classifier is:

$$G(x) = sign(\sum_{m=1}^{M} \alpha_m G_m(x))$$

Where G_m is the mth classifier and α_m is the weight associated with that classifier.

Support Vector Machine

The fourth and final model we used was a Support Vector Machine with a Radial Basis Function kernel. The reasoning for this is that we wanted to test out one more non-linear classification model that was not built on top of decision trees. We thought that a natural choice would be to use an SVM with a non-linear kernel, we decided to use a Radial Basis Function (rbf) kernel.

The Support Vector Machine is fairly intuitive. It attempts to construct a maximum margin hyperplane that separates the different classes. Within this there are a few parameters that w can adjust. The first, and often most important is the kernel that is being used. Changing the kernel allows us to change our features from a linear domain to a non-linear one. This called the "kernel trick" and it is especially useful if our data is not linearly seperable. In our situation, we used the rbf kernel. The equation for the rbf kernel for two input vectors is: $K(x_i, x_j) = e^{-\gamma(x_i - x_j)^2}$ where γ is the kernel coefficient. Intuitively, one can think about it as the inverse of our similarity distance, meaning a smaller γ value allows points that are further apart to be considered similar. The second important parameter of the SVM is the regularization parameter (C). This parameter essentially "tells" the SVM model how much you want to avoid miss classifying each training example. A large value of C will consider a smaller margin hyperplane. This is important because having a more robust margin that allows for some miss classification can help prevent overfitting.

So, it should come as no surprise that the most relevant parameters to test are the gamma value for our RBF kernel (γ), and the regularization parameter (C). After applying the cross-validation function, we found the best values to

be $C = 1000$ and $\gamma = 0.001$.

We can see the loss function for a Support Vector Machine with a RBF kernel as follows:

$$J(\theta) = C[\sum_{i=1}^{m} y_i max(0, 1 - \theta^T \Phi(x_i, x)) + (1 - y_i) max(0, 1 + \theta^T \Phi(x_i, x)]$$

where C is the regularization parameter and $\Phi(x_i, x)$ is the kernel function.

3.5 Model Selection and Comparison of Results

Initially my partner and I wanted to perform model selection using both the oversampled and undersampled data. However, when running the CVparams function to find the best parameters of each model using GridSearchCV, we soon realized that this was not feasible given our current limitations. The entire GridSearch process took about 8 hours for AdaBoost and it was going on for over 10 hours for RandomForest before we decided to cut it short. We think that this is fair given that the only hardware we have to train our models are our laptops. Additionally, and interestingly, we tested out some random configurations of parameters for both AdaBoost and RandomForest being trained on the oversampled dataset and we found that the test performance of each was significantly lower than the models we trained on the undersampled data.

The parameter and model selection process was detailed in section 3.2. The algorithm used to find the best parameters can be seen in Algorithm 1 and the process can be seen in Figure 1. Following is a table of the best models, their parameters, the mean validation score of the training data split, and the model's performance on the separate validation set.

Table 2: Model Selection Table

Model	Parameters	Average CV Score	Val. Score
L_1 Logistic Regression	$\lambda = 0.001$	0.719	0.671
L_2 Logistic Regression	$\lambda = 10$	0.719	0.662
AdaBoost	$lr = 0.1$, $n\_estimators = 200$	0.760	0.724
RandomForest	$max\_depth = 60$,n_estimators=500	0.777	0.742
SVM	$C = 100$, $\gamma = 0.001$	0.741	0.676

From observing Table 2 one can see that the best performing model is the Random Forest model, followed closely by the AdaBoost model. Taking the RandomForest model with the best parameters and applying it to the test set we get a final test score of 0.743. It is most likely no surprise that our best results came from models based off of decision trees. This is advantageous for complex data such as this where the data is not very linearly separable. Additionally, the random subset technique used in the model prevents over fitting as well as the decision boundaries being able to also fit very complex data. We also had a lot of features which is great for our tree based methods as they have best feature selection baked in.

4 Final Results

As previously stated the best overall model was a RandomForest model with parameters of $max\_depth = 100$ and $n\_estimators = 2000$. We can estimate the out of sample error, by using our performance on the test set.

$$E_{out}(h_g) \leq E_{\mathcal{D}_{Test}}(h_g) + \epsilon_M$$

Where:

$$\epsilon_M = \sqrt{\frac{1}{2N_{Test}} ln(\frac{2M}{\delta})}$$

Here:

$$N_{Test} = 45560,\ M = 1,\ \delta = 0.05,\ E_{\mathcal{D}_{Test}}(h_g) = 0.743$$

Thus:

$$E_{out}(h_g) \leq 0.743 + \sqrt{\frac{1}{2 * 45560} ln(\frac{2 * 1}{0.05})}$$

So:

$$E_{out}(h_g) \leq 0.749$$

5 Contribution of each Team Member

Both team members contributed equally to all aspects not mentioned in the specific areas. On top of that, we were both continuously collaborating with one another while we were working on our respective sections of the project.

Contributions of Armin

Armin developed the two feature importance techniques as well as the wrapper function for the iterative imputer class from sklearn (algorithm 2). Along with that he made the GridSearchCV wrapper function (algorithm 1) to do parameter selection for each of the models on the undersampled data.

Contributions of Cameron

Cameron did all of the work reading in all the different datasets. He took care of reading in the datasets into their own respective dataframes, cleaning up the dataframes by pruning irrelvant information, and combining all of the different datasets based off the DATETIME object. He also provided insight on how the the models performed with respect to the oversampled data.

6 Summary and Conclusions

The key finding of our paper was that we could predict with around 74% accuracy, given date/time information and weather conditions, whether there will be a fire or not. This is promising and reassuring as it is much better than chance.

Something that really surprised us was how poorly the models trained on the oversampled data performed relative to the models trained on the undersampled data. Perhaps, if we had more compute resources and more time, we would be able to test out more parameters and get to the bottom of why this is the case. Unfortunately, within the scope of this project, that was simply just not feasible for us.

As far as future work is concerned. We think that it would be interesting to maybe try and include more fire data next time, perhaps from different datasets that contain fire information. Obviously this approach would bring with it it's own problems about matching data from different distributions, but it would still be interesting to test out nonetheless.

7 References

[1] "Climate Change: Global Temperature: NOAA Climate.gov." Climate Change: Global Temperature — NOAA Climate.gov, 14 Aug. 2020, www.climate.gov/news-features/understanding-climate/climate-change-global-temperature.

[2] The Wildland Fire Assesment System (WFAS), www.wfas.net/.

8 Appendix

Please find the file named "main.py". This file will output the validiation set results of our 5 different models. Each one was pre-trained on our training dataset and the model contains the best parameters as found from our CVparams function (Algorithm 1). Additionally, the file will output the test results of our best model, which in this case was the RandomForest model.

The data folder contains the csv file of our unsplit dataframe. The dataframe is then split into training, validation, and test inside of "main.py". We kept the random state the same too so the results should be the same.