Supplement: Algorithmic Details

1. Introduction

All machine learning algorithms used in this paper and described in this supplement are, at least as applied here, supervised learning models used for the purpose of classification. The prediction problem here is said to be supervised because each predictive model is being trained and tuned on labeled data, that is, historical data from which the outcome-observed FRCs-are known. In the case of FRC forecasting, a finite number of possible observations exist: VFR, MVFR, IFR, and LIFR conditions. Because the forecast problem involves predicting a discrete category rather than a quantified, numeric predictand, the machine learning task is deemed to be a classification problem rather than a regression problem. As a broad overview, each of these supervised predictive models ingests as input numerous labeled training examples and uses these to train a final predictive model, which serves as the output of the training phase of this process. Specifically, the outputted, trained model ingests one or more unlabeled examples and outputs a prediction—either a single best-guess classification or assigns a verifying probability to each possible classification category—for the true label of each input example. Aside from the label, each training example also possesses with it a representation of the information available on which to make a prediction. This is typically formatted as a list of predictors, or features, which altogether comprise a feature vector. For this forecasting application, the features are forecast variables from the GEFS/R model, with each individual feature corresponding to a specific atmospheric field forecast at a given latitude and longitude, from a specific ensemble member, depending on what ensemble information is being used to train the predictive model. The length of the feature vector may be expressed as the product of the number of GEFS/R atmospheric fields, the number of grid points from which each field is sampled, and the number of ensemble member values being extracted: 

\[ N_{\text{features}} = N_{\text{fields}} \times (2r + 1)^2 \times N_{\text{ens.mem}} \]

The next section describes in more detail the inner-workings of each employed algorithm, including elaboration on how each model is trained. Section 3 discusses how these algorithms were tuned in the context of this study, and section 4 presents the results of the tuning process.

2. Descriptions of Algorithms

2a. Decision Trees

Decision trees are one fairly basic method for approaching classification problems. Decision trees, for the purposes of this study, consist of a network of two types of nodes: decision nodes and leaf nodes. Decision nodes each have exactly two children, which may be either decision nodes or leaf nodes, with a binary split based on the numeric value of a single feature from the an input example’s feature vector. A leaf node has no children and instead, makes a categorical prediction of the verifying class of the input example based on the leaf’s relationship to its ancestor nodes. For a given input example, one always begins at a decision tree’s root, and at each decision node, compares the value of its feature to the critical threshold of the corresponding feature prescribed for that decision node. If the example’s feature exceeds the node’s critical threshold, the tree is traversed to the node’s right child; otherwise, the tree is traversed to the left child. This process is repeated until a leaf node is reached; at this point,
the value corresponding to the leaf becomes the predicted verifying category for the input example. In this way, the predictive model acts to make a categorical prediction by means of a conjunction of boolean variables derived from an example’s feature vector. Once a decision tree is built, determining a prediction given a feature vector is rather straightforward; the challenge comes in the training phase in constructing the tree.

The two primary questions that must be addressed in constructing a decision tree are:

1a) At a given juncture, how is it determined what feature to split on?

1b) After determining a splitting feature, what determines the critical threshold?

2) What determines when to stop node splitting, and thus create a leaf node?

Suppose a decision tree is trained on \( n \) training examples, each with a feature vector \( F \) of length \( m \). At a given node \( k \), the candidate splits \( S \) consist of a feature \( f \) and threshold \( \theta \), \( S = (f, \theta) \). The set of training examples that traverse the developing tree to reach \( k \) is denoted \( Q \). \( S \) partitions \( Q \) into \( Q_{left} \) and \( Q_{right} \) by:

\[
Q_{left} = \{(F, y) \in F_f | f < \theta; Q_{right} = \{(F, y) \in F_f | f \geq \theta.\}
\]

There is said to be impurity \( I \) at \( k \) based on \( S \); that is given by

\[
I(Q, S) = \frac{\text{len}(Q_{left})}{\text{len}(Q)}H(Q_{left}(S)) + \frac{\text{len}(Q_{right})}{\text{len}(Q)}H(Q_{right}(S)),
\]

where \( H \) is the impurity function. Among the candidate splits \( S \) at \( k \), the chosen split \( S^* \) is the split satisfying:

\[
S^* = \text{argmin}_S(I(Q, S)).
\]

This process of greedy split selection is continued recursively until the termination criterion is satisfied.

Traditionally, the termination criterion is simply that a node \( k \) is a decision node unless \( \text{len}(Q) = 1 \), in which case a leaf node with prediction \( y_0 \) (\( Q = (F_0, y_0) \)). However, recursing this deep is very susceptible to fitting the noise of the training data, thereby overfitting the predictive model and degrading its generalized skill. To alleviate this concern, often more liberal termination criterion are applied, such as creating a leaf node whenever \( \text{len}(Q) \leq \text{len}_{min}; \text{len}_{min} > 1 \), or by imposing a maximum allowable depth \( D \) of the tree \( \text{depth}(k) \leq D \).

2b. Random Forest Classifier (RAND_FOR)

Decision trees can be a powerful approach for a wide array of applications, but they also have several significant drawbacks. First, they are widely regarded as low bias, high variance solutions. That is, minimal error is introduced by erroneous or oversimplistic assumptions in the model formulation, but the model formulation is very sensitive to the input data upon which it trains, which results in large error when extrapolating to other test data. More succinctly, decision trees are very prone to overfitting the training data, fitting to the noise of the training data rather than just the underlying relationships. This flaw substantially diminishes the utility of decision trees as a general predictive model. Second, the decision tree framework does not robustly extend to a probabilistic framework, since leaf nodes make deterministic predictions based on the mode verifying category of the subset of the training data reaching each respective node; applying a probabilistic prediction to individual leaf nodes greatly compounds the overfitting problem. It has been demonstrated that using many different decision trees to form, in aggregate, a predictive model can significantly decrease the model variance
with only a slight increase to the model bias, provided the trees are sufficiently uncorrelated. This is the idea behind random forests.

The challenge with random forests is: how do you generate a large set (forest) of reasonably skillful decision trees that are not strongly correlated? The procedure described above for generating a decision tree from training data is deterministic, that is, a given set of training data will always produce the same decision tree via that algorithm. A forest of identical decision trees adds no value over using a single decision tree. The extra process for random forest generation is twofold: tree bagging and feature bagging. To generate a forest of size B from the n training examples, tree bagging involves the application of a simple bootstrapping procedure. Specifically, one samples, with replacement, n training examples from the original set, and uses this derived set to construct a decision tree using the method described above. This process is repeated B times to form a forest. Overfitting due to correlated trees can still occur under this approach if a small subset of the original feature space are much more robust predictors of the verifying category than the rest. To overcome this problem, only a random subset of the m original input features are considered at each decision node; the size of the random subset is denoted here as Z; 1 ≤ Z ≤ m.

2c. Gradient Boosting Classification (GRAD_BOOST)

The basic concept of boosting is that a large ensemble of weak learners - very high bias, very low variance models - can form a strong learner. Highly truncated decision trees - discriminating on only a small subset of candidate features and possessing a small number of decision nodes - are a popular choice of weak learner for boosting, and were selected as the ensemble members for this study. Decision trees may be thought of as partitioning the m-dimensional feature space $R^m$ into different segments, and then assigning a verifying category to each fragment based on the mode verifying category of the training data in that subspace. Each decision tree b can thus be characterized by its basis, or predictive, function $h: h_b(F) = \sum_{j=1}^{J} FRC_{bj}X_{bj}$, where $X_{bj} = \begin{cases} 1 & F \in R_{bj} \\ 0 & F \notin R_{bj} \end{cases}$, where J is the number of segmented regions of feature space and $R_{bj}$ is the j’th segment of feature space for the b’th tree; F here is the feature vector, which specifies a location in feature space. The net model $M$ can then be expressed as a weighted sum of the basis functions:

$$M(x) = \sum_{b=1}^{B} h_b(F)\gamma_b,$$

where $\gamma_b$’s are coefficients.

At any step $b-1$, the $b^{th}$ tree is constructed so as to minimize the loss function $L$ satisfying:

$$M_b(F) = M_{b-1}(F) + \arg\min_h \sum_{i=1}^{n} L(y_i, M_{b-1}(F_i) - h(F))$$

In gradient boosting, this minimization problem is accomplished by gradient descent:

$$M_b(F) = M_{b-1}(F) + \gamma_b \sum_{i=1}^{n} \nabla_M L(y_i, M_{b-1}(F_i))$$
Where \( y_i \) corresponds to the verifying category of the \( i \)th training example, and with:

\[
\gamma_b = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, M_{b-1}(F_i)) - \gamma \frac{\partial L(y_i, M_{b-1}(F_i))}{\partial M_{b-1}(F_i)}
\]

For this study, the chosen loss function for probabilistic FRC classification was \textit{multinomial deviance}.

This process is repeated \( B \) times to form an ensemble of size \( B \). Lastly, two extensions of this procedure attempting to reduce the \textit{variance} of the final ensemble are explored in this study. The first is a \textit{learning rate} where the iterative model avoids over-adjusting to new members by applying a dampening coefficient \( \nu \): \( M_b(F) = M_{b-1}(F) + \nu \gamma_b h_b(F) \). The second approach is taken from the idea of random forest creation: use only a subset of the total training data for each new decision tree. Instead of creating a new sample of size \( n \), sampled with replacement from the original dataset, however, \( \alpha n \)-where \( \alpha \) is the \textit{subsampling coefficient} between 0 and 1-training examples are sampled, without replacement, from the original training data.

2d. K-Nearest Neighbors (KNN)

Applying straightforward clustering techniques for classification problems can prove highly effective despite its simplicity. Perhaps the best known, the K-nearest neighbors clustering algorithm is explored here. KNN and other clustering algorithms have the unique property, compared with the other machine learning algorithms discussed here, that it is \textit{non-generalizing}; test example predictions are made purely based on the proximity to training examples, rather than applying a fitted model which is extrapolated based on the training data. This is very advantageous when decision boundaries are highly erratic and non-linear, as other methods will tend to produce \textit{biased} solutions in these instances. However, its inability to identify patterns in the training data can also cause it to use training data less efficiently than other algorithms in many instances.

K-Nearest Neighbors classification makes prediction based on a weighted vote of the K training examples judged most similar to the test example. Similarity of two data points is determined by a distance metric \( D \) applied to the points’ feature vectors \( F_1 \) and \( F_2 \), smaller distances being more similar. The most commonly used distance metric is also the most intuitive, the Euclidean Distance:

\[
D_{\text{Euclid}}(F_1, F_2) = \sqrt{\sum_{i=1}^{m} (F_{1i} - F_{2i})^2}
\]

Distances between the test example and all training examples are computed, and the training examples associated with the K smallest computed distances comprise the set of voting neighbors. Each neighbor votes in accordance with its associated verifying observation in the training data to yield a set of votes, or predictions, \( V \). The final prediction of the KNN algorithm is then the product of matrix \( V \) and a normalized weights vector \( W \). Traditionally, \( W \)'s elements are all \( 1/K \), so that each member has an equal vote, but may be chosen to instead vary with weights inversely proportional to distance.
2e. Support Vector Classification (SVC)

Despite being rather abstract and difficult to interpret, both in the formulation and the output, SVC is an extremely powerful method which presents numerous advantages. Due to its versatility, it generally extends to high dimensional feature spaces better than the other algorithms employed in this study, and can still work effectively even when the dimensionality of the feature space is larger than the number of training examples. Aside from the difficulty in physically interpreting the output of the fitted model, the primary drawbacks of this approach are that it cannot directly solve a multi-class classification problem, and also cannot directly assign probabilities to its predictions. These limitations suggest at first glance that this method may be a poor choice for the problem of probabilistic FRC forecasting, but due the power of the technique in addition to available workarounds, SVC is still examined here.

Support vector machines (SVMs) aim to define the hyperplane(s) which separates the training examples according to their respective labels and maintains as large of a margin as possible from any training example so as to minimize generalization error. Consider a two class problem, where, without loss of generality, all training examples can be associated with either class A, with a value of 1, or class B, with a value of -1. The observation vector Y is thus comprised of elements $y_i \in \{1, -1\}$. Any hyperplane in the $m$-dimensional feature space can be described by: $\mathbf{n} \cdot \mathbf{F} - b = 0$, where $b$ is a scalar and $\mathbf{n}$ is a vector normal to the hyperplane. In the event that the training data are linearly separable in the feature space, then a set of two hyperplanes may be considered: $\mathbf{n} \cdot \mathbf{F} - b = 1$ and $\mathbf{n} \cdot \mathbf{F} - b = -1$; these planes correspond to the nearest boundaries corresponding to each class. As stated above, SVMs are maximum-margin classifiers, that is, they seek to maximize the margin, or distance, between these two bounding planes. It can be readily shown that the margin between these hyperplanes may be expressed as: $\frac{2}{\|\mathbf{n}\|}$, where $\|\mathbf{n}\|$ denotes the norm of the vector defining the hyperplane. Thus, to maximize the separation margin, $\|\mathbf{n}\|$ must be minimized, subject to the constraint that no training example is misclassified. This can be readily expressed as an optimization problem:

Minimize $\|\mathbf{n}\|$ subject to: $y_i (\mathbf{n} \cdot \mathbf{F}_i - b) \geq 1$, $\forall i \in \{1, ..., n\}$

This problem can be readily solved; the implementation details will not be discussed here.

The approach above works well for training data that is linearly separable in feature space, but in general, this is not the case. The SVC approach may be generalized to allow for misclassifications; this is accomplished by creating a slack vector $\Xi$, whose elements $\xi_i$ allow misclassification by changing the constraints to the minimization problem to:

Minimize $\|\mathbf{n}\| + C \sum_{i=1}^{n} \xi_i$ subject to: $y_i (\mathbf{n} \cdot \mathbf{F}_i - b) \geq 1 - \xi_i$, for $\forall i \in \{1, ..., n\}$

In the expression above, $C$ corresponds to the penalty term, or inverse regularization coefficient; it determines how smooth the decision surface should be, with a low value implying a highly regularized, low variance, high bias solution with smooth classification boundaries, while a high value implies a high
variance, low bias solution that attempts to classify all of the training examples as they are actually labeled.

Even the extension above only allows for linear classification: the hyperplane must be defined as a linear combination of the original features. However, in many problems, a non-linear decision boundary better captures the true relationships between the input features and true classifications. However, this limitation can be solved too by use of kernels and the kernel trick. The mathematics of kernel theory and the kernel trick in particular are interesting, but not fundamental to an elementary understanding of SVC and thus will not be discussed here. Succinctly stated, the kernel trick exploits the fact that for some non-linear transformation φ to a feature vector F, \( \varphi(F) \), the inner product of two such transformed vectors \( F_i \) and \( F_j \) may be expressed by a kernel \( k \):

\[
k(F_i, F_j) = \varphi(F_i) \cdot \varphi(F_j).
\]

This can be applied to transform the data into a much higher dimensional space, sometimes even infinite-dimensional, where the optimal decision boundary is linear in the transformed space. Applying this transformation, the problem formulation stays the same, except the kernel function replaces the inner product in the optimization constraints. Many different choices of kernels here; some popular choices that are explored in this paper are: 1) Linear \( k(\vec{a}, \vec{b}) = \vec{a} \cdot \vec{b} \), 2) Polynomial of degree \( d \) \( k(\vec{a}, \vec{b}) = (\gamma(\vec{a} \cdot \vec{b}) + r)^d \), 3) Radial basis function (RBF) \( k(\vec{a}, \vec{b}) = e^{-\gamma \|\vec{a} - \vec{b}\|^2} \), and 4) Sigmoid \( k(\vec{a}, \vec{b}) = \tanh(\gamma(\vec{a} \cdot \vec{b}) + r) \).

The final limitations, namely (1) inability to extend to multi-class problems, and (2) inability to extend to probabilistic output, present more genuine problems in that they don’t have ‘pure’ solutions. The former limitation has numerous possible workarounds. The approach utilized in this study applies a “one-versus-one” approach where \( n_{\text{classes}}(n_{\text{classes}}-1)/2 \) classifiers are fit to the training data, with each classifier corresponds to a unique pair of classification labels. The aggregate of classifiers is then used to make final class predictions. Probability estimates are made using a version of Platt Scaling; the method as applied here is both quite esoteric and ad hoc. As such, the method does have some known theoretical issues; principally, the predicted class in the deterministic problem may not have the plurality of the probability assignment in the probabilistic output. The details of the probability assignment phase will not be discussed here; for more information, see the Sci-Kit Learn User’s Guide.

3. Methods: Parameter Tuning

Each algorithm has multiple parameters to tune, and a subset of those parameters were selected for explicit tuning for each machine learning algorithm examined in this study. RAND_FOR (Breiman 2001) can be tuned on the number of decision trees in the forest \( B \) in section 2, the criterion for node splitting- Gini impurity or entropy/information gain- \( H \) in section 2, and several other tree specific parameters that were not tuned in this study. GRAD_BOOST (Friedman 2001) was similarly tuned on the number of decision tree estimators \( B \), the maximum allowable depth of a single estimator \( D_{\text{MAX}} \), the number of considered features/predictors to consider in determining the optimal node split \( \alpha_{\text{features}} \), and the fraction of total training data to use when fitting an individual estimator \( \alpha_{\text{subsample}} \). KNN was tuned on number of nearest neighbors (observation records) \( K \) to consider in generating forecast probabilities (FPs), the metric \( D \) used to determine the distance between two observations (Euclidean or
Mahalanobis), and the metric \( W \) used to weight the nearest neighbors in probability generation (uniform weights or distance-proportional). Lastly, SVC (Cortes and Vapnik 1995; Wu et al. 2004) configurations explored the choice of kernel function \( k \) (linear, quadratic, cubic, sigmoid, radial basis function (RBF)), and various combinations of the kernel coefficient, \( \gamma \), and the regularization coefficient, \( C \). Several options were also available for calibrating the probabilities output by the trained statistical models; the algorithms tested were: 1) no additional probability calibration, 2) parametric sigmoid-based probability calibration (Platt 1999), and 3) non-parametric isotonic probability calibration (Niculescu-Mizil and Caruana 2005).

4. Results: Parameter Tuning by Cross-Validation

The KNN classifier was tuned first on the number of neighbors, \( K \). 100 neighbors produced the highest cross-validation ARPSS of those tested (not shown); this was found to be a fairly sensitive parameter, with skill scores dropping substantially at low neighbor counts. Euclidean \( (D = D_{euc}) \) rather than Mahalanobis distance was found to yield better results, and uniform weighting \( (W = <1/K, 1/K, \ldots, 1/K>) \) rather than distance based weighting to form FPs from the identified nearest neighbors was also found to be superior. Tuning in this parameter space was not exhaustive, and better-scoring parameters may well exist. The RAND_FOR classifier was tuned on the number of trees and on the metric for determining node splits. A forest size, \( B \), of 1000 trees was found to produce the maximum skill score; however, this was also the largest forest size analyzed, as computation became quite expensive for significantly larger forest sizes. Increasing the forest size did appear to have a significant impact, particularly at small forest sizes. Entropy was found to be superior to Gini impurity for determining node splits (\( H \)), but this difference was quite small. GRAD_BOOST was also tuned; only decision trees were examined as the boosted classifier. Decision stumps \( (D_{max} = 1) \) were found to produce superior results to trees with higher depths, and depth was considered to be a moderately sensitive parameter. Again, 1000 estimators \( (B = 1000) \) was found to produce the best results, and this was the largest number tested. \( (\alpha_{features}, \alpha_{subsample}) \) space, where \( \alpha_{features} \) is the fraction of features to examine in node splitting decisions and \( \alpha_{subsample} \) is the fraction of the training data to consider for training of an individual estimator, was then tuned. Reducing these values tends to yield more biased, lower variance solutions. Optimal parameters were found to be \( (0.5, 1.0) \), but neither appeared to be particularly sensitive in the region of parameter space considered. The SVC model was tuned first on choice of kernel. The RBF kernel was found to produce the best cross-validation ARPSS; the sensitivity (not shown) was found to be generally order 0.01, with the sigmoid kernel performing significantly worse, at least for the \( (C, \gamma) \) pair analyzed, which was the default \( (1, 1/3402) \), with 3402 being the number of model features \( m \). After the RBF kernel was selected, \( (C, \gamma) \) space was explored to determine the optimal setting. The \( \gamma \) coefficient corresponds to the radius of influence of a training example; with lower values having a larger radius of influence. The optimal parameters were found to be \( (10, 1/3402) \) in the parameter space studied. For none of the algorithms studied did subsequent probability calibration from isotonic or Platt methods produce improved cross-validation results.