Paper ML15

A Comparison of Machine Learning Algorithms for Binary Classification of Free-text Fields Using SAS, R, and Python

Michael S Rimler, GlaxoSmithKline, Cincinnati, Ohio, US
James Vasquez, GlaxoSmithKline, Collegeville, Pennsylvania, US
Allison Hainline, GlaxoSmithKline, Collegeville, Pennsylvania, US

ABSTRACT

Clinical data collected via free-text fields are often difficult to use in analysis due to spelling or grammatical errors on data entry by humans, or simply stemming from differences in languages across global sites. Monitoring the data and querying sites to clean the content to support analyses is time consuming, even if the site is responsive and agreeable. Downstream processing of these data in preparation for analyses supporting clinical study reporting can also be quite challenging.

This paper compares various methodologies for developing a binary classifier on free-text data. Machine learning methods in SAS, R, and Python are compared to an exact string search in SAS developed for 100% accuracy on the training dataset. A comparison of results of applying the trained models on the test dataset is presented, both individually and as combined methodologies.

SECTION 1: BACKGROUND

Analyzing text data via machine learning algorithms is a common activity in the data science arena. Applications include scanning social media posts to identify signals of imminent criminal behavior or using converted voice responses to navigate a customer through an automated phone menu. People have even compiled publicly available databases of text data for building and testing models, such as `thesimpsons` which contains script lines for approximately 600 episodes of the cartoon series **The Simpsons** [1] and the `schrute` package containing the entire transcripts from the American sitcom **The Office** [2].

In our context, we focus on a specific application of text analytics: binary classification of free-text data. The example uses collected information on concomitant medication indications to identify whether a record was administered to treat COPD (chronic obstructive pulmonary disease) or not. A common method for this type of classification in the clinical reporting process leverages an external spreadsheet containing all medication records collected in a study, manually reviewed to be flagged COPD or not, and then merged back into the database during the analysis dataset production process. Rimler and Pitlyk [3] illustrated that using supervised machine learning (ML) algorithms for the binary classification problem can perform at least as well as brute force string searching and enhance efficiency relative to a manual review process. This paper discusses the implementation of three supervised ML classification algorithms (logistic regression, random forests, and naïve Bayes classifiers) across three commonly used programming languages (SAS, R, and Python).

The objective of this exposition is to provide the reader with insight and core syntax on the implementation each algorithm in each programming language, including highlighting language-specific challenges that may be encountered. As with any problem addressed by data science methodologies, the explicit and optimal solution implemented is heavily dependent on the problem statement, the data available, and the ultimate objective (e.g., a deployable product). In this paper, we focus on the aspects that are generically applicable to the binary classification of text data, instead of distracting the reader with bespoke syntax that is specific to our application.

In general, we do not perform any hyperparameter tuning to optimize the performance of each individual algorithm. We simply want to start with data, prepare it for ingestion by the supervised ML algorithms, and produce results. While we attempt to compare similar implementations across languages, our primary objective is to simply illustrate the language specific ML method implementation. Where possible, we will maintain the Precision-Recall tradeoff objective of Rimler and Pitlyk [3], buying higher recall (less false negatives) at the expense of lower precision (more false positives). For more detailed information regarding tuning hyperparameters of the implemented functions/procedures, please refer to the source documentation for each algorithm included in the Appendix. For example, we have provided links to all included Python functions from the `scikit-learn` package.

SOURCE DATA

The data used for the analysis is drawn from GSK's R&D Information Platform (RDIP) which houses pooled and anonymized clinical trial data and is the same data that supported the analyses in Rimler and Pitlyk [3]. From the subset of data extracted from RDIP, and after additional data cleaning, the resulting analysis dataset includes 212,079 unredacted records across 106 studies and 18,481 unique values of CMINDC. In preparation for the analyses, the records were manually labeled as COPD or not. The only difference between the data used in this paper and that which supported previous analyses is the correction of 3 mislabeled records in the training dataset (discussed in Rimler and Pitlyk [3]).

Table 1 contains a description of the values of CMINDC within the analysis dataset, many of which appear more than once. Approximately 35.7% of the full data are labeled as indicated for COPD (i.e., the 'truth'), compared to only 3.1% of the unique terms.

Table 1. Descriptive Statistics of CMINDC within the Analysis Dataset

	All Values (N=212,079)
Labeled as COPD medication	75,623 (35.7%)
Unique terms	18,481
Labeled as COPD medication*	564 (3.1%)

** Percentage is calculated using number of unique terms as the denominator.*

Among the unique terms, 2.3% contain 'COPD' as an exact string with 10 not labeled as COPD. For example, a CMINDC value of 'NON COPD' or 'PANIC ATTACK SECONDARY TO COPD' is not labeled as COPD. In addition, 41 unique records contain at least one of the following strings, treated as an exact proxy for COPD.

- CHRONIC OBSTRUCTIVE PULMONARY DISEASE
- CHRONIC BRONCHITIS
- EMPHYSEMA

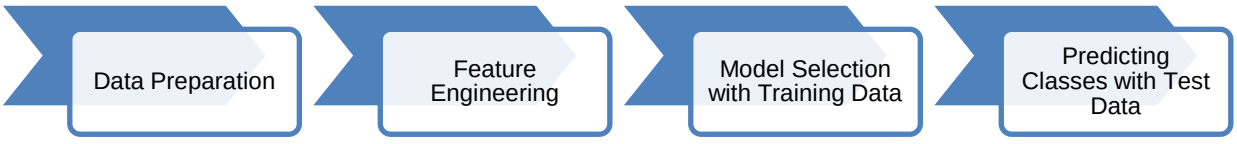
Six of these 41 records also contain the 'COPD' string. For similar reasons to the string 'COPD', not all records containing a proxy string are labeled as COPD.

Approximately 20% (113 / 564) of the remaining unique records labeled as COPD contain misspellings of one of these four proxies, such as 'COIPD', 'CHRINIC OBSTUCTIVE PULMONARY DISEASE', 'EMPHYSIMIA', and 'CHRONIC BRONHITIS'. Correct classification of these records as COPD is the ultimate objective of methods illustrated in this paper, since exact string searching becomes a bespoke problem in each application.

SECTION 2: BINARY CLASSIFICATION OF FREE-TEXT DATA

In this paper, we follow a general set of steps in the development of any of the classifiers. This process is not unique to our problem and is likely to be more complex when the objective is an optimized classifier ready for deployment. The process is depicted in Figure 1.

Figure 1: General Process Flow Implemented for Binary Text Classifier Development



SECTION 2.1: DATA PREPARATION

STRATIFICATION INTO TRAINING AND TEST SETS

For all methods and languages, the analysis dataset is stratified into two groups using Python's `train_test_split` from `scikit-learn`'s `model_selection`. After stratification, we have two datasets: a training dataset which contains 80% of the records (169,663) and a test dataset which contains the remaining 20%. From this point, we retain only unique values of CMINDC. The training dataset contains 15,988 such unique values, of which 495 (3.1%) are labeled as COPD (see Table 2). The stratification method works well, as the overall training dataset has 60,498 (36.1%) records labeled as COPD, consistent with the combined train-test data. In the interest of full disclosure, we use the

same stratification from Rimler and Pitlyk [3] and simply corrected the mislabeling of 3 records from COPD to not COPD without re-stratification, explaining the slight imbalance in COPD labeled records between the training and test datasets. This will not impact the purpose of the exposition presented in the paper.

Table 2. Unique Terms in Train vs Test Datasets

	Training Set (N=169,663)	**Test Set (N=42,416)**
Labeled as COPD medication	60,498 (36.1%)	15,125 (35.7%)
Unique terms	15,988	6,143
Unique terms labeled as COPD medication*	495 (3.1%)	195 (3.2%)
Unique COPD labeled records appearing in only one dataset	369 / 495 (74.5%)	69 / 195 (35.4%)

** Percentage is calculated using number of unique terms as the denominator.*

The test dataset contains 69 additional unique values of CMINDC that are labeled COPD and do not exist in the training dataset (see Table 2). Therefore, approximately 12.2% of our global list of unique labeled COPD indications are not available to the training dataset (but will be included in the test phase). All algorithms discussed in this paper are calibrated on the training dataset prior to running on the test dataset.

SECTION 2.2: FEATURE ENGINEERING

Preparing the data for analysis is nearly the same for both the training and test datasets. One necessary step is to lowercase the indication field so that the underlying data is not case sensitive. The rest of the data preparation is aimed toward *feature engineering*, a process which creates characteristics of the data that can be leveraged for predictive modeling.

TF-IDF

In our analyses, the only features we create are TF-IDF weightings based on the training dataset. The TF in TF-IDF means *term frequency* and represents the number of times that the term (word) appears in each record. IDF means *inverse document frequency* and measures the incidence of records in the entire database which contain the word. Mathematically combining the number of occurrences of a word in a record (TF) with a measure of the number of records which contain that word in the overall dataset (IDF) produces a normalized score of how important that word is in that record. We do not discuss the explicit transformation here and instead refer the reader to documentation such as Scikit-Learn. [4] TF-IDF weightings are the only features used in the algorithms for this paper.

ADDITIONAL FEATURE ENGINEERING

For the practitioner, depending on the available data, many other features could be considered in an attempt to improve the classification capability of the underlying data. Rimler and Pitlyk [3] included ***edit distance measures*** of a record to each of the proxy terms for COPD such as 'EMPHYSEMA'. An edit distance measure is a metric that characterizes how different two text strings are from one another. For example, 'michael' is very close to 'micheal', relative to 'james' or 'allison'. Another common method for feature engineering in text analytics is referred to as ***n-grams***. This method looks at incidences of word -tuples, such as word pairs or word triples, in a similar way as we have considered word counts. Your particular application for binary classification of free-text and input data may allow more opportunities for feature engineering, potentially improving the performance of the trained model.

SECTION 2.3: MODEL SELECTION WITH TRAINING DATA

For any given classification algorithm, there are typically many characteristics of the function/model executed which the practitioner can manipulate in order to find the 'best' calibrated model. In the machine learning world, these characteristics are referred to as *hyperparameters* and the practitioner engages in *hyperparameter tuning* using the training data to select the best model.

During this process, cross-validation can be implemented to reduce the risk of *over-fitting*. One type of cross-validation is called *k-fold* cross-validation. In this method, the training data is partitioned into *k* folds (e.g., 5 folds), the model is fit on *(k-1)* folds and tested on the last fold. This process is repeated *k* times, giving each fold an opportunity to serve as the testing fold. Model fit criteria are aggregated (e.g., averaged) and the aggregate model fit statistic gives a comparative measure of model performance against other hyperparameter profiles. Another cross-validation method is called 'leave-one-out' and is essentially an extreme case of *k*-fold cross-validation each fold has exactly one observation.

Although we do not explicitly engage in hyperparameter tuning in this exercise, we are interested in a comparative static measure of the precision-recall tradeoff as the threshold for classifying a predicted probability value as COPD

or not. For example, the default may assign any predicted probability greater than 0.5 as COPD. Lowering this threshold to 0.4 would have the impact of flagging more records as COPD. This would increase false positives and, perhaps, reduce false negatives. Therefore, we would be increasing recall at the cost of reduced precision. This comparative static exercise is of interest and using a cross-validation technique could provide more robust measures of precision and recall at each threshold value.

LOGISTIC REGRESSION CLASSIFIERS

The first classification we will discuss is the **logistic regression** classifier. As with the other two methods we will discuss, logistic regression is a supervised method in that the classifier is trained on pre-labeled data, i.e. the truth is known. The features which are created from the data are used as explanatory variables in the model to estimate the probability that a given record should be classified as COPD or not. Therefore, our context applies **binomial** logistic regression for the classification process.

RANDOM FOREST CLASSIFIERS

Classification using the **random forest** method combines a number of decision tree methods to produce a score that is ultimately used to classify a record into one of the target classes. Decision tree methods may fall prey to overfitting and the random forest methodology aims to correct for the overfitting tendency.

NAÏVE BAYES CLASSIFIERS

Classification using the naïve Bayes classification method leverages Bayes' theorem to generate a probabilistic score using naïve independence assumptions between the features of the data. In our case, we implement **multinomial naïve Bayes**. In this application, for each feature, the value in the feature vector (TF-IDF scoring, in our case) represents a histogram of the incidence of each feature on the records. The observed incidence, coupled with the independence assumption, permits the generation of the likelihood of a particular histogram being observed. This can be mapped into a scoring algorithm of the likelihood that a record's feature profile should be classified as COPD.

SECTION 2.4: PREDICTING CLASSES WITH TEST DATA

Once the model has been selected (calibrated) for a given classifier, we can feed the test data into the fitted model and assess the performance of its class prediction via common measures such as accuracy, precision, and recall. The test data must be prepared similar to the training data, including any engineered features such as TF-IDF weightings. Specifically for TF-IDF, since we use a dictionary of words derived from the training data, it is possible that new vocabulary words appear in the test data and receive zero weight when input into the classifier. Note that a new free-text concomitant indication value could still receive TF-IDF weights. But if a new word existed within that new indication value, it would receive zero weight. Only if a new free-text value contained zero vocabulary words would that record receive no TF-IDF weight. The modeler should consider engineering additional features from the training data, if it is desired to reduce this possibility. For example, in Rimler and Pitlyk [3], inclusion of edit distances allow for a new test data value of 'CAPD' to get some merit due to its proximity to 'COPD'.

After calculating the predicted probability of COPD classification using the selected threshold (e.g., probability above 0.5 is classified as COPD), performance statistics of the classification method such as accuracy, precision, and recall can be generated and analyzed.

SECTION 3: PYTHON IMPLEMENTATION

In Python, we generated TF-IDF vectors in a two-step process. First, we used `CountVectorizer` from Scikit-Learn's `feature_extraction` to look at every unique word available in the (training) dataset and create a variable (vector) for each word and populate the value of that variable with the number of times that word appears in the text field for each record. We then applied `TfidfTransformer` from the same library to normalize the counts based on the TF-IDF algorithm.

```
count_vect = CountVectorizer()
tfidf_transformer = TfidfTransformer()
X_train_counts = count_vect.fit_transform(x_train)
X_train_tfidf = tfidf_transformer.fit_transform(X_train_counts)
```

We can also perform the same data preparation techniques on the test dataset. We need to ensure that the same bag of words and normalization is used on the test data as was on the training data. Therefore, to use the same objects on the new data with these Scikit-Learn modules in Python, the syntax uses the `transform` function instead of the `fit_transform` for the vectorization procedures:

```
X_test_counts = count_vect.transform(x_test)
X_test_tfidf = tfidf_transformer.transform(X_test_counts)
```

For the three classifiers of interest, the code within Python is almost identical. The only difference is loading the proper module and defining the classifier (`clf`):

Logistic Regression

```
from sklearn.linear_model import LogisticRegressionCV
from sklearn.model_selection import cross_validate
clf = LogisticRegressionCV()
```

Random Forest

```
from sklearn.ensemble import RandomForestClassifier
clf = RandomForestClassifier()
```

Naïve Bayes

```
from sklearn.naive_bayes import MultinomialNB
clf = MultinomialNB()
```

For logistic regression, we use `LogisticRegressionCV` from the `linear_model` module. For random forests, we use `RandomForestClassifier` from the `ensemble` module. And for naïve Bayes, we use `MultinomialNB` from the `naive_bayes` module. Since we are not optimizing the algorithms, we simply employ the default parameterization of each respective function. If one were interested in optimizing, then the parameters suppressed in each respective `clf` definition would be updated to modify from their default values.

Recall that our explanatory information has been defined as `X_train_tfidf`. If the true COPD labels in the training set is defined as `y_train`, then we have the following Python code, which employs 5-fold cross-validation using `cross_validate` from the `model_selection`:

```
X = X_train_tfidf
y = y_train
r = cross_validate(clf, X, y, cv=5,
                   scoring=['accuracy', 'precision', 'recall'],
                   return_train_score=True)
```

The parameter `cv` specifies the number of folds used in the cross-validation method. The last two parameters (`scoring` and `return_train_score`) facilitate diagnostics on the executed model and are not required for successful implementation.

DETERMINING A CLASSIFICATION THRESHOLD PROBABILITY

Even though our objective is not to optimize, we do want to establish a threshold probability (score) that determines whether a record is classified as COPD (=1) or not COPD (=0). Our primary interest is to increase recall at the cost of lower precision (i.e., reduce the number of false negatives at the expense of higher false positives). The precision-recall curves for the Python classifiers are presented in Figure 2.

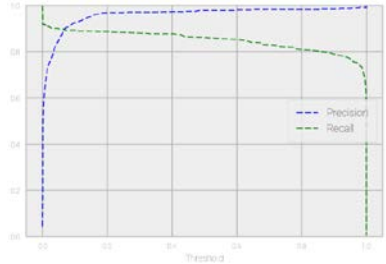
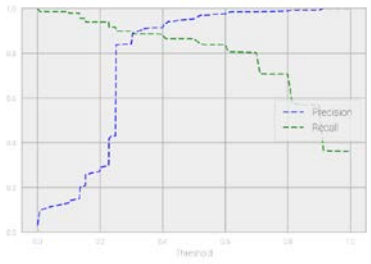
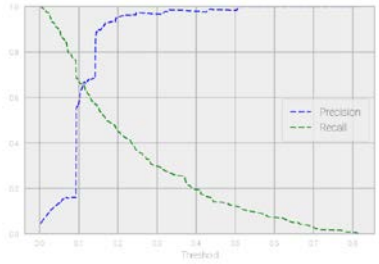
Logistic Regression	Random Forest	Naïve Bayes
		
SELECTED_THRESHOLD = 0.005	SELECTED_THRESHOLD = 0.260	SELECTED_THRESHOLD = 0.094

Figure 2: Precision-Recall Plots for Logistic Regression, Random Forest, and Naïve Bayes Classifiers in Python

For Logistic Regression, a threshold of 0.005 yielded the preferred balance of increasing recall at the expense of lower precision, a desired characteristic in this particular application. For Random Forest, a threshold of 0.260 is selected. Note that these are the same thresholds selected in Rimler and Pitlyk [3]. For Naïve Bayes, the threshold selected is 0.094.

We are now ready to test the trained model on the prepared test data by feeding it into the model and obtaining the predicted probability of COPD classification based on the `SELECTED_THRESHOLD` with each classifier. The Python

code is the same in each classifier, since the difference in classifier is embedded in the definition of `clf`.

```
y_test_pred = clf.predict_proba(X_test_tfidf)
y_test_pred_classes = (y_test_pred[:,1] > SELECTED_THRESHOLD).astype(int)
```

This information can be used to calculate performance statistics of classification method such as accuracy, precision, and recall. These are presented in Table 3 and compared to the results from the 100% Accuracy exact string search presented in Rimler and Pitlyk [3]. Even without optimization, the classifiers perform fairly well. With some optimization effort on the training set, or further feature engineering, performance on the test set may have been improved. Note, however, that good practice does not permit us to engage in this manner due to data leakage. If we were to use these results to tune the model more, we risk overfitting on the test data and reduce the potential effectiveness of a deployed model on newly collected data in future studies.

Table 3: Performance Comparison of Supervised Binary Classifiers on Test Data using Python

Classifier	Threshold	Accuracy	Precision	Recall	False Positives	False Negatives
100% Accuracy String	N/A	0.999	0.993	0.769	0	5
Logistic Regression	0.005	0.989	0.760	0.959	59	8
Random Forest	0.260	0.995	0.910	0.938	18	12
Naïve Bayes	0.094	0.990	0.895	0.790	18	41

CREATING AN ENSEMBLE MODEL

An extension of the analyses above would build an ensemble model of the three individual methodologies. An **ensemble model** combines the results from multiple classification algorithms and applies a decision rule on whether a record should be classified as COPD. The random forest classifier itself is an ensemble model because it combines the results from randomly selected decision trees in its classification process.

We restrict our presentation to Python because the concept is not language specific and one language is sufficient for this purpose. We present results from our application from 5 ensemble models known as **voting classifiers** [5]. The first 3 are pairwise ensembles of two of the classifiers. The decision rule applied labels as COPD if at least one of the 2 models predicts as COPD. This rule will tend to generate more false positives and lower false negatives, which is an underlying objective of this application. The other 2 ensembles incorporate all 3 classifiers: one with the same decision rule (if predicted COPD in at least one classifier, then label as COPD) and the other with a majority rule decision rule (if predicted COPD in a majority of classifiers, then label as COPD). Results are presented in Table 4. More information on additional types of ensemble modeling can be found in Géron (2017) [5].

Table 4: Results from Ensemble Models from Python Implementation

Classification Method	Accuracy	Precision	Recall
Logistic Regression and Random Forest	0.987	0.730	0.959
Logistic Regression and Naïve Bayes	0.988	0.739	0.959
Random Forest and Naïve Bayes	0.993	0.860	0.949
All three classifiers (Rule: At least one COPD)	0.987	0.719	0.959
All three classifiers (Rule: Majority COPD)	0.995	0.907	0.949

One could argue that the majority rule method using all 3 classifiers achieves higher precision without a significant loss of recall or accuracy. However, we argue that these types of ensemble models add little value for this particular problem. The primary reason for this is that the predicted classes for each record do not vary much across the three algorithms. Indeed, of the 6,143 records in the testing dataset, only 116 terms are classified differently in at least one algorithm.

There are 31 records which provide the main variance across the data but comprise about 0.5% of the testing data. Among the other records, 43 are correctly classified as non-COPD in random forest and naïve Bayes, but incorrectly classified as COPD under logistic regression. There are an additional 31 terms correctly classified as COPD by logistic regression and random forest, but falsely classified as non-COPD under naïve Bayes. Three records are false positives in all classifiers and 8 records are false negatives in all classifiers. Therefore, we do not observe much improvement when comparing the ensemble approach to the individual classifiers. It was a worthwhile investigation, but a stand-alone classifier seems to perform just as well as the combined methods.

Our analysis into the ensemble models focused on an *ex post* investigation of results on the testing data. An alternative would investigate the ensemble model *ex ante* on the training data, perhaps adjusting the hyperparameters of the individual models to obtain performance enhancements on the ensemble model. If one were to take this approach, it is advisable to consider the appropriate splitting of data for cross-validation in each stage to ensure proper out-of-sample testing on fitted models in each stage. In addition, optimizing each individual classifier prior to applying to the testing data, with the ultimate ensemble classification in mind, would be more appropriate and mitigate the potential for data leakage (see [6] for a good discussion on the topic of data leakage).

SECTION 4: R IMPLEMENTATION

While there are several R packages that are equipped with the TF-IDF functionality, we have chosen to use the `text2vec` package. The first step separates the phrases into individual words, or "tokens", which are used to create our vocabulary. This is similar to Python's `CountVectorizer`. We then use TF-IDF weighting to transform our vocabulary into a document-term matrix that is normalized by the number of times each token appears in the dictionary. In the code that follows, all functions are available in the `text2vec` package.

```
it_train = itoken(nodupes$Text,
                  preprocessor = tolower,
                  tokenizer = word_tokenizer,
                  ids = nodupes$id,
                  progressbar = FALSE)
vocab = create_vocabulary(it_train)
vectorizer = vocab_vectorizer(vocab)
dtm_train = create_dtm(it_train, vectorizer)
tfidf = TfIdf$new()
X_train_tfidf = fit_transform(dtm_train, tfidf)
```

We may then use the same TF-IDF model to transform our test data, allowing prediction of COPD classification based on a fitted model on the training data set.

```
it_test = itoken(y_test$COPD,
                 preprocessor = prep_fun,
                 tokenizer = tok_fun,
                 ids = y_test$id,
                 progressbar = FALSE)
dtm_test_tfidf = create_dtm(it_test, vectorizer)
X_test_tfidf = transform(dtm_test_tfidf, tfidf)
```

Unlike Python, the three classifiers are computed using different R functions, though many ways exist to do the same fundamental model fitting, these are just the ones we have chosen for ease of use.

For **logistic regression**, we have chosen to use the `glmnet` function from the `glmnet` package. We use the `cv.glmnet` function to perform 5-fold cross validation in order to choose the optimal lambda value to be used for L2 regularization. Note that we have set alpha=0, which corresponds to a ridge regression framework, which aligns with the default behavior of the logistic regression function in Python. We then use the optimal lambda for our final model fit, which is used to predict the labels on the new data.

```
lambda_vals = 10^seq(3, -2, by = -.1)  # Lambda values to search
cv_glm_fit = cv.glmnet(x = X_train_tfidf, y = y_train$COPD,
                       family = 'binomial', alpha = 0, lambda = lambda_vals, nfolds = 5)
opt_lambda = cv_glm_fit$lambda.min
glm_fit = cv_glm_fit$glmnet.fit
pred_class = predict(glm_fit, s = opt_lambda, newx = X_test_tfidf, type='class')
```

For **random forest** implementation in R, we use the `ranger` function/package in partnership with the `train` function in the `caret` package to fit and cross-validate our model. The `ranger` function is preferred for random forest fitting, since it has been shown to have much faster performance on large data sets. We fit the default 500 forests and perform 5-fold cross-validation, which is conducted via the train function.

```
control <- trainControl(method="repeatedcv", number=5, repeats=3, search="random")
                        x=as.matrix(X_train_tfidf)
```

```
COPD_rf <- train(x=as.matrix(X_train_tfidf), y=y_train$COPD, method="ranger",
                 metric='Accurary', tuneLength=15, trControl=control)
```

In this implementation, we use cross-validation to determine the best value for the `mtry` option in ranger, which controls the number of variables to split at in each node. We use the random option in the `trainControl` function, which allows for grid searching in a random fashion along all possible values of the parameter. This option is useful when we have no idea what the plausible range parameter value will be.

Once the model has been trained, we can apply our final model to calculate the predicted classes for the test data using the `predict` function.

```
pred_class <- predict(COPD_rf,  newdata = as.matrix(X_test_tfidf))
```

To implement **naïve Bayes** classification in R, we use the `multinomial_naive_bayes` function from the `naivebayes` package. Note that we have chosen not do use cross-validation for this model, as we are not optimizing any parameters. Cross-validation may be useful in order to optimize parameters, such as a Laplace smoother. The results would likely improve with this optimization.

```
mnb <- multinomial_naive_bayes(x = as.matrix(X_train_tfidf), y = y_train$COPD)
```

Once we have the model fit, we can use it to predict the class labels for the test dataset. We can then impose the threshold on the new predicted probabilities to determine predicted classes for the new data and use these classes to calculate our desired performance statistics.

```
pred_prob <- predict(mnb, newdata = as.matrix(X_test_tfidf), type = "class")
```

This information can be used to calculate performance statistics of classification method such as accuracy, precision, and recall. These are presented in Table 5 and compared to the results from the 100% Accuracy exact string search presented in Rimler and Pitlyk [3].

Table 5: Performance Comparison of Supervised Binary Classifiers on Test Data using R

Classifier	Threshold	Accuracy	Precision	Recall	False Positives	False Negatives
100% Accuracy String	N/A	0.999	0.993	0.769	0	5
Logistic Regression	0.5	0.994	0.924	0.872	14	25
Random Forest	0.5	0.997	0.978	0.938	4	12
Naïve Bayes	0.5	0.971	0.525	0.954	168	9

SECTION 5: SAS IMPLEMENTATION

We did not find a SAS distributed procedure for calculating the TF-IDF weight features with our data, therefore we needed to develop original code to tokenize and derive the weights. There may exist some community developed open SAS code, but we decided to derive it ourselves. In what follows, we provide snippets of code that illustrate the steps described. There may be some non-critical intermediate steps that are omitted from the discussion and code presentation.

The first step, similar to Python, is to create the dictionary of words from the set of unique terms in the training data and count the incidence of each word in each record, with each word as a separate variable in the SAS dataset. After defining ` ` as the number of spaces in each indication record (i.e., the number of words – 1), we can separate each word into individual variables with:

```
data DS2(drop=i);
   set DS1;
   length wd1-wd%eval( .+1) $200;
   array words(%eval( .+1)) $ wd1-wd%eval( .+1);
   i=1;
   do until (scan(cmindc,i," ") eq "");
      words(i)=scan(cmindc,i," ");
      i+1;
   end;
run;
```

Transposing this from wide to long and removing duplicate words, we obtain the dictionary of unique words (or vocabulary) from the training data:

```
proc transpose data=DS2 out=DS3(drop=_name_);
   by type cmindc copdyn copdidx;
   var wd1-wd%eval( .+1);
run;
proc sort data=DS3 out=Dict(keep=col1) nodupkey;
   by col1;
run;
```

We then give a unique index to each word in the dictionary, which allows for tracking each word through the process, for example transposing the dictionary back to wide:

```
data Dict2;
   set Dict;
   by col1;
   txtid = _n_;
   name  = "txt"||strip(put(txtid,best.));
run;
proc transpose data=Dict2 out=Dict3(drop=_name_);
   var col1;
   id name;
run;
```

We then combine this record back to each record in the training dataset:

```
proc sql noprint;
   create table DS4 as
   select *
   from DS2, Dict3;
quit;
```

Finally, we count the incidence of each word within each record, where `&txtid` is the number of words in the dictionary :

```
data DS5;
   set DS4;
   array words(%eval( .+1)) $ wd1-wd%eval( .+1);
   array terms(%eval(&txtid.))     txt1-txt%eval(&txtid.);
   array flags(%eval(&txtid.))     flg1-flg%eval(&txtid.);
   do kk = 1 to %eval(&txtid.); ** Initialize counts**;
      flags(kk) = 0;
   end;
   do ii = 1 to %eval( .+1);
      do jj = 1 to %eval(&txtid.);
         if words(ii) = terms(jj) then flags(jj) = flags(jj) + 1;
      end;
   end;
   mby=1; * Used for merge key later *;
run;
```

Now that we have the word counts (term frequency), the next step for TF-IDF weights is to calculate the inverse document frequency (IDF). We follow the steps detailed in Liu's description. [7] The first step is to count how many records in the training dataset contain term. We execute this via retain statement, a loop, and keeping the last record only:

```
data DS6(keep=idf:);
   set DS5 end=eof;
   retain idf1-idf%eval(&txtid.) 0;
   array flags(%eval(&txtid.)) flg1-flg%eval(&txtid.);
   array idfs(%eval(&txtid.))  idf1-idf%eval(&txtid.);
   do jj = 1 to %eval(&txtid.);
      *** Count number of documents a word appears *;
      if flags(jj) > 0 then idfs(jj) + 1;  * If word appears, increase by 1 *;
      else idfs(jj) = idfs(jj);            * else keep same *;
   end;
   if eof;  * Keep last record (total num docs) *;
run;
```

The next step calculates the IDF for each word by

IDF = log(Number of Records / (1 + Number of documents with the word))

using code such as:

```
data DS7(drop=jj);
   set DS6;
   array idfs(%eval(&txtid.))  idf1-idf%eval(&txtid.);
   do jj = 1 to %eval(&txtid.);
      idfs(jj) = log(%eval(&numdocs.)/(1+idfs(jj)));
   end;
   mby=1; * Used for merge key later *;
run;
```

Finally, we merge back to the training data and use L2 normalization described by Liu [7] to normalize each IDF within each record:

```
data DS8;
   merge DS5 DS7;
   by mby; * Artificial merge key *;
   array flags(%eval(&txtid.)) flg1-flg%eval(&txtid.);
   array idfs(%eval(&txtid.))  idf1-idf%eval(&txtid.);
   array tfidfs(%eval(&txtid.))  tfidf1-tfidf%eval(&txtid.);
   array sqs(%eval(&txtid.))  sq1-sq%eval(&txtid.);

   do jj = 1 to %eval(&txtid.);
      tfidfs(jj) = flags(jj)*idfs(jj);*Product of counts and inverse doc freq*;
      sqs(jj) = tfidfs(jj)**2;        *Squares*;
   end;

   sumsq = sqrt(sum(of sq1-sq%eval(&txtid.))); *Sum of squares within record *;

   do jj = 1 to %eval(&txtid.);
      tfidfs(jj) = tfidfs(jj) / sumsq;  *Normalization by sum of squares     *;
   end;
run;
```

We now have the dataset DS8 which contains 9288 variables representing the 9288 unique words from the training dataset and containing a value equal to that word's TF-IDF weight in that record. For the logistic regression classifier, we can use PROC LOGISTIC:

```
proc logistic data=DS8;
   class tfidf1-tfidf%eval(&txtid.);
   model copdyn(event='1')= tfidf1-tfidf%eval(&txtid.) / RIDGING = NONE;
   score data=testdata out=predprobs;
run;
quit;
```

Here, in theory, the procedure will fit the model with the DS8 dataset and score the test data via the **score** statement. There are a few things to note about this specification of the model:

- We are not using any cross-validation techniques in this implementation. PROC LOGISTIC does have functionality for cross-validation, but it implements the 'leave-one-out' method. In addition, if we were to manually create a k-fold cross validation procedure, at the time of publication, we could not find a consensus on how to 'combine' the k predicted models into one predictive model for the test data.
- We have selected the option `RIDGING = NONE` based on notes to the log. If we were focusing on optimizing the algorithms, this decision would face more thorough scrutiny.
- The test dataset (testdata) must be prepared the same as the training data. This means that it also has 9288 additional variables, each representing a word in the training data dictionary. We use the same IDF values to calculate the TF-IDF. Therefore, as previously discussed, completely new vocabulary words in the test data will receive zero TF-IDF weight and yield no explanatory/fit power.
- Even with all of this, we were unable to successfully execute the procedure due to memory limitations. The Hessian becomes quite large and our instance of SAS could not handle the memory requirements. And even if Mamore wasn't a limiting factor, extrapolating out based on execution with a subset of the features, we expect the model would take at least 24h to converge (vs. a few minutes in Python).

For the reader's interest, SAS also offers PROC HPLOGISTIC within SAS High-Performance Statistics, which is intended to exhibit higher performance. But, in our limited experience, we did not find HPLOGISTIC any more attractive than LOGISTIC in our application, relative to Scikit-Learn in Python.

Due to the computational requirements, we were unable to produce results using PROC HPFOREST (for random forest classification) or for naïve Bayes. Indeed, each classifier seems only available via SAS Enterprise Miner [8] or publicly available code such as provided by Pliner (2017). [9]

Therefore, we are unable to present any classification results using SAS based procedures at the time of publication.

SECTION 6: CONCLUSION

The objective of this paper was to compare different classification algorithms on free-text data across popular programming languages of Python, R, and SAS. Sample code for feature engineering, fitting the model, cross-validation, and predicting on test data for each language were presented and discussed where feasible.

The main findings across the languages are as follows:

- **Python** is relatively easy to implement in all stages. Code is similar across classifiers. Computational performance is fast, relatively.
- **R** might be a good first step for R programmers, but otherwise we suggest Python given our experience in this project. The sheer number of fitting functions to choose from serves as both a pro and con for R. It allows a great deal of flexibility for the experienced programmer but may be overly complicated for a beginner. Computational performance tends to be slower in R than in Python, requiring vectorization and other techniques to increase speed, and may intimidate the inexperienced.
- **SAS** is time consuming, not as easy to code, maybe not well suited for this application. In reduced data execution, SAS proved to consume both time and memory which was ultimately debilitating on a full run. Additionally, our implementation using SAS was limited by what was available to us with our standard SAS installation. Moving forward, leveraging the capabilities of SAS Enterprise Miner may prove more productive in achieving predicted classes on the test data.

In addition, if one were to extend this analysis for a real-world application, then the practitioner would need to consider optimizing the algorithms in the training phase. This may include further feature engineering, perhaps similar to Rimler and Pitlyk [3]. It would also surely involve hyperparameter tuning of the available parameters of each function/procedure, depending on the implementation language. Python's Scikit-learn actually includes some grid search capabilities. We refer the reader to the package's documentation referenced in the appendix. Finally, there may be other ensemble models, as discussed in Géron (2017) which may yield improved performance on the optimized classifiers [5]. After the final, optimized classifier is developed and trained on the training data, whether an ensemble model or not, the practitioner would need to deploy the classifier to production on newly collected free-text data, a step that is beyond the scope of this paper.

APPENDIX: DOCUMENTATION OF PACKAGES AND PROCEDURES

Language	Utility / Function Online Documentation Source
Python	**`scikit-learn`** package https://scikit-learn.org/stable/
	`train_test_split` from `sklearn.model_selection` https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.train_test_split.html
	`CountVectorizer` from `sklearn.feature_extraction` https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.CountVectorizer.html
	`TfidfTransformer` from `sklearn.feature_extraction` https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfTransformer.html
	`cross_validate` from `sklearn.model_selection` https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.cross_validate.html
	`LogisticRegressionCV` from `sklearn.linear_model` https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html
	`RandomForestClassifier` from `sklearn.ensemble` https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html
	`MultinomialNB` from `sklearn.naive_bayes` https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.MultinomialNB.html

Language	Utility / Function Online Documentation Source
R	**text2vec** package https://cran.r-project.org/web/packages/text2vec/index.html
	ranger package https://cran.r-project.org/web/packages/ranger/index.html
	e1071 package https://cran.r-project.org/web/packages/e1071/index.html
	glmnet package https://cran.r-project.org/web/packages/glmnet/index.html
	caret package https://cran.r-project.org/web/packages/caret/index.html
	naivebayes package https://cran.r-project.org/web/packages/naivebayes/index.html
SAS	**PROC LOGISTIC** http://support.sas.com/documentation/cdl/en/statug/68162/HTML/default/viewer.htm#statug_logistic_syntax.htm
	PROC HPLOGISTIC https://support.sas.com/documentation/onlinedoc/stat/151/hplogistic.pdf
	PROC HPFOREST https://documentation.sas.com/?docsetId=emhpprcref&docsetTarget=emhpprcref_hpforest_toc.htm&docsetVersion=15.1
	PROC HPBNET https://documentation.sas.com/?docsetId=emhpprcref&docsetTarget=emhpprcref_hpbnet_toc.htm&docsetVersion=15.1&locale=en

REFERENCES

[1] The Simpsons dataset: https://github.com/jcrodriguez1989/thesimpsons

[2] The schrute package: https://www.rdocumentation.org/packages/schrute/versions/0.1.0

[3] Rimler, Michael and Matt Pitlyk (2019). “Performing Analytics on Free-Text Data Fields: A Programer's Wurst Nitemare.”, PharmaSUG 2019, Paper BP-079. https://www.pharmasug.org/proceedings/2019/BP/PharmaSUG-2019-BP-079.pdf

[4] Scikit Learn package: https://scikit-learn.org/stable/

[5] Géron, Aurélien (2017). *Hands-On Machine Learning with Scikit-Learn & TensorFlow*. O’Reilly Media, Inc.

[6] “How Data Leakage Impacts Machine Learning Models” Last accessed: January 16, 2020. https://mlinproduction.com/data-leakage/

[7] Liu, Ethen. *TF-IDF, Term Frequency-Inverse Document Frequency*. https://ethen8181.github.io/machine-learning/clustering_old/tf_idf/tf_idf.html Last accessed: January 28, 2020.

[8] Liu, Ye, Weihua Shi, and Wendy Czika (2017). “Building Bayesian Network Classifiers Using the HPBNET Procedure.” https://support.sas.com/resources/papers/proceedings17/SAS0474-2017.pdf Last accessed: January 28, 2020.

[9] Pliner, Vadim (2017). “A SAS® Macro for Naïve Bayes Classification.” https://www.lexjansen.com/nesug/nesug04/po/po09.pdf Last accessed: January 28, 2020.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Michael S Rimler	James Vasquez	Allison Hainline
michael.s.rimler@gsk.com	james.x.vasquez@gsk.com	allison.e.hainline@gsk.com

GlaxoSmithKline
1250 S. Collegeville Road
Collegeville, Pennsylvania, US, 19426-0989