Ravirajpurohit

Apr 30 · 19 min read · Listen

Reddit Post Classifier

Link to the Jupyter Notebook

Link to the dataset

Brief about the problem

This time, I will work with a dataset of Reddit posts containing information about the posts and respective comments as a separate file. The goal is to predict which subreddit category a post belongs to using the information available and individual comments.

We will be looking at multiple ways to approach this problem and analyze which method is best.

About the dataset

There are three categories of target variable subreddit — MachineLearning, datascience, and artificial. The dataset is a CSV file "Top_Posts.csv" containing the following columns —

- post_id : The unique identifier of the post or comment
- post_title : The title of the post
- post_url : The URL link to the post
- subreddit : the subreddit where it was posted (target label)
- flair_text : The user-assigned tag to the post
- score : The net score (upvotes minus downvotes) of the post
- comments : The number o[illegible] ...e post

- upvote_ratio : The ratio of upvotes to total votes
- date-time : The date (Y-M-D) and time (in UTC) when the post was created
- year : The year when the post was created

There is another dataset "Top_Posts_Comments.csv" containing the post_id and various comments for each post.

Data Exploration

The very first step for any machine learning problem is data exploration. The data scientist should be completely familiar with the kind of data they are dealing with; kind of features, missing data, the data type of features, and meanings of features.

Upon looking at the columns available to us in the dataset, we can identify some features which are either hard to handle or meaningless to process. Two such columns are — post_url and date-time.

```
## load the dataset; remove the useless columns
dataset = pd.read_csv('data/Top_Posts.csv').drop(columns=['post_url','date-time'
dataset.head()
```

- We should look at the target variable to find out how many categories are there to handle; if it is going to be a binary classification or multiclass classification problem.

```
## check how many items in each class
dataset.subreddit.value_counts()
```

```
artificial        999
MachineLearning   998
datascience       990
Name: subreddit, dtype: int64
```

- Check the dataset’s information to find out how many nulls exist and what is the data type of each column. This will define the data cleaning and preprocessing steps.

```
## check each column's information
dataset.info()
```

```
RangeIndex: 2987 entries, 0 to 2986
Data columns (total 8 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   post_id       2987 non-null   object
 1   post_title    2987 non-null   object
 2   subreddit     2987 non-null   object
 3   flair_text    2441 non-null   object
 4   score         2987 non-null   int64
 5   comments      2987 non-null   int64
 6   upvote_ratio  2987 non-null   float64
 7   year          2987 non-null   int64
dtypes: float64(1), int64(3), object(4)
memory usage: 186.8+ KB
```

- The dataset description also helps do the same as the previous step. It gives the view into the data.

Merge the two datasets to create one single Dataframe to work with. We can merge using post_id as the row identifier as it is the common column between the two datasets.

```
## import another dataset containing comments for each post
dataset_add = pd.read_csv('data/Top_Posts_Comments.csv')
dataset_add.head()

## add comments to the main dataset and merge the two
dataset = dataset.merge(right=dataset_add.groupby(by='post_id', as_index=False).
                        how='inner',
                        on='post_id')
```

Data Preprocessing

The plan is to vectorize all the vocabulary available for each category and identify the target label using the vocabulary for available comments.

Since the post title column also contains some information about the post content, we can combine it with the comment column to include those words in the vocabulary.

```
data = dataset.copy()
data['comment'] = data.apply(lambda row: row['post_title']+
                             ' '+
                             ' '.join(np.array(row['comment']).astype(str)),
                             axis=1)
data.drop(columns=['post_title'], inplace=True)
```

Tokenize Vocabulary

To create the word vocabulary, we need to clean the text available in the dataset. Basic cleaning involved steps like — removing newline character, stripping empty spaces, getting rid of "", and remove non-alphabetical words.

We will also lemmatize the vocabulary to wipe the words with similar meanings and context.

```
lemmatizer = WordNetLemmatizer()

def preprocess(text):
    """
    steps like -
    converts all characters to lower
    removes non-alphabetical words

    Parameters
    ----------
    text - string

    Returns
    -------
    text - string (preprocessed)
    """
    text = text.lower()
    text = text.replace('"','').replace('\n','').strip()
    ## word_tokenize splits a sentence into words linguistically
    text = nltk.word_tokenize(text)
    text = [word for word in text if (word.isalpha() and word not in stopwords.w

    word_list = [lemmatizer.lemmatize(word) for word in text]
    return ' '.join(word_list)

data['comment'] = data['comment'].apply(preprocess)
```

TF-IDF

The very first algorithm that we're going to explore is the famous TF-IDF i.e. Term Frequency-Inverse Document Frequency algorithm along with the Multinomial Naive Bayes algorithm.

It is a handy algorithm that uses the frequency of words to determine how relevant those words are to a given document. It calculates a score for each word based on its frequency in a document and its rarity in the overall corpus. This algorithm is widely used for information retrieval, document classification, and text-mining tasks.

The main advantage of using TF-IDF over the Naive Bayes classifier for text classification is that it can help to improve the accuracy of the classification by weighing the features based on their relevance. TF-IDF takes into account the frequency of the words in the document and the rarity of the words in the overall corpus, which can help to identify important features and reduce the impact of common features that are not informative for classification. Naive Bayes, on the other hand, does not consider the relevance of the features and assumes that all features are equally important.

Experiment 1. A — Use Comment as the only feature

Let's try working with only the textual data available in the feature Comment. We have to define our target variable as an integer to easily process it. We can use factorize method for this.

```
X = data.comment
y, y_org = data.subreddit.factorize()
```

The factorize method converts the categorical label into integers and saves the order in *y_org* variable.

Now we can follow the classic machine learning steps and divide the dataset X,y into training, testing, and validation datasets.

> Quick Question — Why do we need a validation dataset?
>
> *In machine learning, the main goal is to build a model that can generalize well to new and unseen data. To achieve this, we typically split the available data into three sets: training, validation, and testing.*
>
> *The training set is used to train the model parameters, and the testing set is used to evaluate the performance of the trained model on new and unseen data. The testing set should be used only once, after the model is fully trained, to avoid overfitting.*
>
> *The validation set is used to tune the hyperparameters of the model, such as the learning rate, regularization strength, or the number of hidden units in a neural network. These hyperparameters cannot be learned from the data but need to be set manually by the user. By evaluating the model performance on the validation set, the user can select the best hyperparameters that result in the highest accuracy or lowest error on the validation set.*
>
> *Using a separate validation set instead of tuning the hyperparameters on the testing set ensures that the testing set remains unbiased and provides a fair estimate of the model's generalization performance. It also allows the user to compare different models and select the best one based on the performance of the testing set.*

- Split the dataset into Training, Testing, and Validation splits.

```
## split the dataset into train-test-valid stratified split;
## to keep the ratio of classes same
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.20,
                                                    random_state=42,
                                                    stratify=y)

X_train, X_val, y_train, y_val = train_test_split(X_train,
                                                  y_train,
                                                  test_size=0.10,
                                                  random_state=42,
                                                  stratify=y_train)
```

- Let us define a pipeline to vectorize the text data and apply the Multinomial Naive Bayes algorithm.

```
## define the pipeline
text_clf = Pipeline([('tfidf', TfidfVectorizer()), ('clf', MultinomialNB())])

## fit the classifier
text_clf.fit(X_train, y_train)
```

- We can test the performance of the classifier on the three datasets.

```
y_pred_train = text_clf.predict(X_train)
y_pred_test = text_clf.predict(X_test)
y_pred_val = text_clf.predict(X_val)

print(f'Train Accuracy - {round(100*accuracy_score(y_train, y_pred_train),2)}%')
print(f'Test Accuracy - {round(100*accuracy_score(y_test, y_pred_test),2)}%')
print(f'Validation Accuracy - {round(100*accuracy_score(y_val, y_pred_val),2)}%\

print(f'Classification Report - \n{classification_report(y_test, y_pred_test)}')
```

```
Train Accuracy - 66.96%
Test Accuracy - 60.8%
Validation Accuracy - 64.02%

Classification Report -
              precision    recall  f1-score   support

           0       0.50      0.81      0.62       200
           1       0.74      0.96      0.84       198
           2       1.00      0.05      0.09       199

    accuracy                           0.61       597
   macro avg       0.74      0.61      0.51       597
weighted avg       0.74      0.61      0.51       597
```

We can look at the classification report to analyze the performance of the model. This model provides us with the following observations -

Observations -

The classifier shows moderate accuracy i.e. around 60–65%, for these particular data samples. The classification report further clarifies the model performance. Here, the output labels are -

- 0 — MachineLearning
- 1 — datascience
- 2 — artificial

The classification report provides an evaluation of the classifier's performance for each class, as well as an overall evaluation of its accuracy, precision, recall, and F1-score. The overall model accuracy is okay and consistent among train, test, and validation datasets. But looking at the classification report, we come to know that the model is not performing any good for predicting posts for the "artificial" category using this dataset. The best performance is for predicting "datascience" posts, where we see 0.84 F1-score, which is pretty good.

The macro-average and weighted-average F1-scores are relatively low, also indicating that the classifier's overall performance is not very good, and it is struggling to correctly classify the posts.

Overall, these results suggest that the classifier could benefit from further improvements, such as using a more sophisticated algorithm or improving the quality of the dataset and the preprocessing steps.

Keep record of each model for later comparison -

Let us save the model configuration and the performance in a dictionary to analyze at the end of all the experiment

```
perf_comp['title'].append('TF-IDF NB - Comment')
perf_comp['train_acc'].append(round(100*accuracy_score(y_train, y_pred_train),2)
perf_comp['test_acc'].append(round(100*accuracy_score(y_test, y_pred_test),2))
perf_comp['valid_acc'].append(round(100*accuracy_score(y_val, y_pred_val),2))
perf_comp['clf'].append(text_clf)
```

Experiment 1. B — Use flair_text feature

Looking at the dataset, we can see that there is one more column that seems to contain the relevant information that can identify the category of the Reddit post. Let us explore this column and try to use this as a machine-learning feature.

Before planning to use a feature for prediction, we should look at the number of missing values or presence of NaNs as we did before.

```
print(f'{round(100*data.flair_text.isna().sum()/data.shape[0],2)}% of column is
```

```
18.31% of column is NaN
```

Usually, when we have missing data, we can fill using techniques like most frequent. But since the percentage of NaN in this feature is significant, hence we should not fill these instances.

Rather, let's work with the rest of the data.

```
X = data.flair_text.dropna().astype(str).apply(preprocess)
y, y_org = data[~data.flair_text.isna()].subreddit.factorize()
```

We might notice, this step brings us back to the same approach. We have to divide this data into train, test, and validation datasets to train the model.

```
## split the dataset into train-test-valid stratified split; to keep the ratio o
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.15,
                                                    random_state=42,
                                                    stratify=y)

X_train, X_val, y_train, y_val = train_test_split(X_train,
                                                  y_train,
                                                  test_size=0.10,
                                                  random_state=42,
                                                  stratify=y_train)
```

```
X_train.shape, y_train.shape
```

Fit the model using the same pipeline as earlier and find the prediction results. The output is —

```
Train Accuracy - 58.56%
Test Accuracy - 63.11%
Validation Accuracy - 61.84%

Classification Report -
              precision    recall  f1-score   support

           0       0.59      0.56      0.57       138
           1       0.65      0.83      0.73       130
           2       0.68      0.47      0.55        98

    accuracy                           0.63       366
   macro avg       0.64      0.62      0.62       366
weighted avg       0.63      0.63      0.62       366
```

> *Observations -*

Again, the classifier shows moderate accuracy i.e. around 63%, for these particular data samples. The classification report further clarifies the model performance. Here, the output labels are -

- 0 — MachineLearning
- 1 — datascience
- 2 — artificial

The classification report provides an evaluation of the classifier's performance for each class, as well as an overall evaluation of its accuracy, precision, recall, and F1-score. The overall model accuracy is okay and consistent among train, test, and validation datasets.

Improvement -

Looking at the classification report, we see an improvement in the performance as the model is pretty consistent among all three categories. The best performance is again for predicting "datascience" posts, where we see 0.83 F1-score, which is pretty good.

The macro-average and weighted-average F1-scores are relatively low but more balance than before, indicating that there has been improvement in classifier's overall performance.

However, there is still a good scope for improvement.

The observation here is, that the keywords in the flair_text column are significant as well to balance the model performance.

Random Forest Classifier

Random Forest is a popular machine-learning algorithm that is well-suited for classification problems like the one at hand. It is an ensemble learning method that builds multiple decision trees and combines them to make the final predictions. In the case of a classification problem, each decision tree in the random forest is trained on a random subset of the data, and the final prediction is made by taking a majority vote from the individual trees.

Let's try the Random Forest method for this dataset.

We will do this in the exact same way that we trained the previous classifier. The only difference is to replace the Multinomial Naive Bayes algorithm with Random Forest Classifier algorithm.

I will only mention the steps which are different from the previous approach. Please refer to the jupyter notebook for details of code for each experiment.

```
## pipeline for model
text_clf = Pipeline([('tfidf', TfidfVectorizer()),
                     ('clf', RandomForestClassifier(n_estimators=100,
                                                    random_state=42))])
```

The output of the classification report is -

```
Train Accuracy - 100.0%
Test Accuracy - 81.03%
Validation Accuracy - 81.1%

Classification Report -
              precision    recall  f1-score   support

           0       0.83      0.77      0.80       150
           1       0.91      0.75      0.82       149
           2       0.73      0.91      0.81       149

    accuracy                           0.81       448
   macro avg       0.82      0.81      0.81       448
weighted avg       0.82      0.81      0.81       448
```

Observations

This time, the classifier shows good accuracy i.e. 81%, for these particular data samples. The classification report further clarifies the model performance. Here, the output labels are -

- 0 — MachineLearning
- 1 — datascience
- 2 — artificial

The classification report provides an evaluation of the classifier's performance for each class, as well as an overall evaluation of its accuracy, precision, recall, and F1-score. The overall model accuracy is okay and consistent among train, test, and validation datasets.

Improvement -

Looking at the classification report, we see an improvement in the performance as the model is pretty consistent among all three categories. In this case, the model is performing equally well for all three classes and the overall accuracy is pretty as well.

However, there seems to be an overfit as there is a significant gap in training and testing accuracy. This can be improved by hyper-parameter tuning.

Hyperparameter Tuning

After tuning hyperparameters, I found the following to be the best configuration showing a balanced output.

```
## pipeline for model
text_clf = Pipeline([('tfidf', TfidfVectorizer()),
                     ('clf', RandomForestClassifier(n_estimators=100,
                                                    random_state=42,
                                                    criterion='gini',
                                                    max_depth=10,
                                                    min_samples_split=50,
                                                    min_samples_leaf=10))])
```

Output is -

```
Train Accuracy - 81.89%
Test Accuracy - 78.12%
Validation Accuracy - 77.56%

Classification Report -
              precision    recall  f1-score   support

           0       0.79      0.74      0.77       150
           1       0.94      0.71      0.81       149
           2       0.68      0.89      0.77       149

    accuracy                           0.78       448
   macro avg       0.80      0.78      0.78       448
weighted avg       0.80      0.78      0.78       448
```

As we can see, the overfitting has been reduced without compromising on the testing and validation accuracy.

Experiment 2. B — Use Comment as the feature

Similarly, we can try to work with flair_text and use a Random Forest classifier for classification in the pipeline. Let us see how that works out.

Since we have done this before and can arrange the code changes, let us look at the output for this approach.

```
Train Accuracy - 64.68%
Test Accuracy - 67.49%
```

```
Validation Accuracy - 66.67%

Classification Report -
              precision    recall  f1-score   support

           0       0.67      0.56      0.61       138
           1       0.68      0.95      0.79       130
           2       0.68      0.47      0.55        98

    accuracy                           0.67       366
   macro avg       0.67      0.66      0.65       366
weighted avg       0.67      0.67      0.66       366
```

Observations

We can see that the model performance is balanced, but the overall accuracy is Low. Though the model is not overfitting, so there is no need to tune the hyper-parameters like earlier.

However, the model configuration can be tweaked to increase the overall accuracy.

But before that, I believe one of the observations that we discussed earlier is made clearer. That is, the "comment" feature and "flair_text" feature are not the same, even though for Multinomial Naive Bayes, the latter feature was performing significantly better.

The most significant parameter is choosing the model, which is making the most difference in the overall accuracy of the model.

Experiment 2. C — Use non-textual features

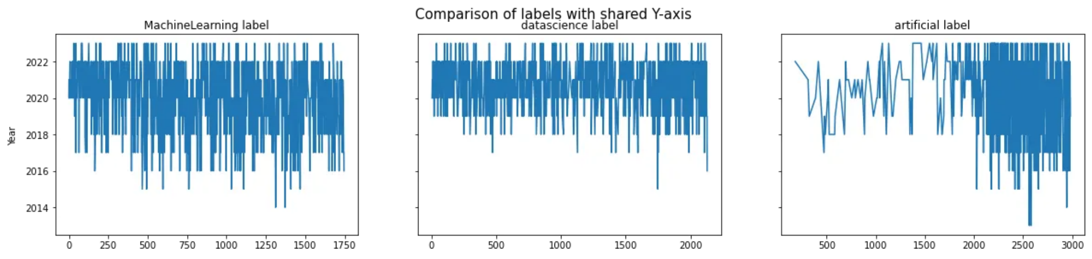
Even though the initial goal was to build a text classifier, we can see that we have other features at our disposal which may be relevant in determining the category of Reddit posts.

hypothesis

A very simple and solid idea is to explore features using visuals. We have 3 categories of data points and a few non-textual features. We can plot graphs to see if the data points are easily separable in two dimensions or not.

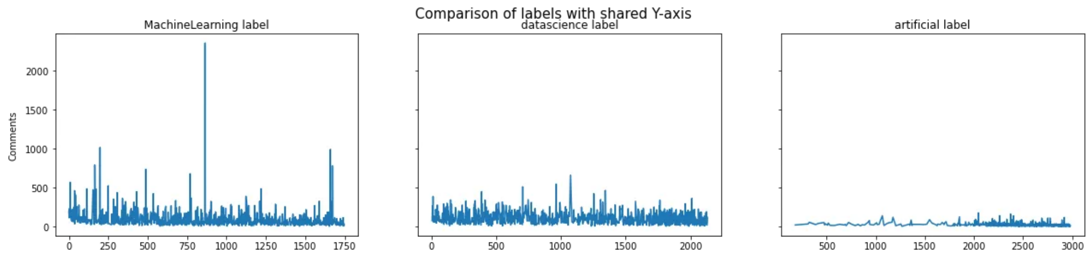
This helps in defining a few hyper-parameters like which model to use, which kernel to use in the case of SVM etc. Let us observe the features in plots and formulate a hypothesis.

```
fig, axs = plt.subplots(1,3,sharey=True,figsize=(20,4))

## year
axs[0].plot(data[data.subreddit=='MachineLearning'].year)
axs[0].set_title('MachineLearning label')

axs[1].plot(data[data.subreddit=='datascience'].year)
axs[1].set_title('datascience label')

axs[2].plot(data[data.subreddit=='artificial'].year)
axs[2].set_title('artificial label')

fig.suptitle('Comparison of labels with shared Y-axis', fontsize=15)
fig.text(0.09, 0.5, 'Year', va='center', rotation='vertical')

plt.show()
```

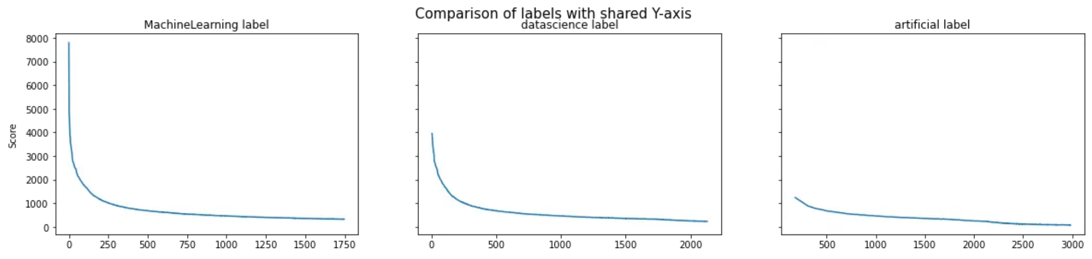


Visualize label separation with Year feature

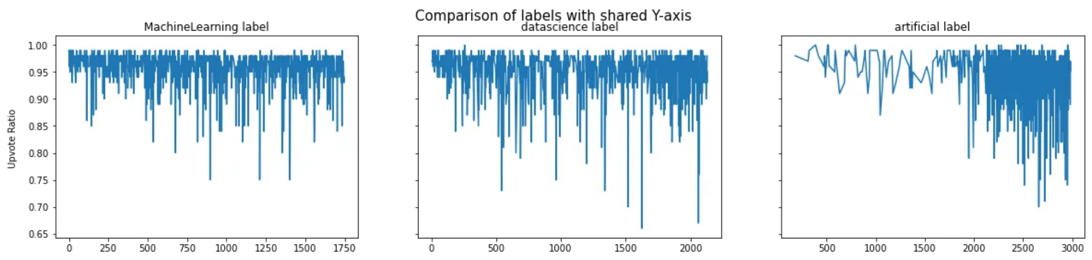
Similarly, we can plot and visualize the target variables with different numerical features and a shared Y-axis.



Visualize label separation with Comments feature



Visualize label separation with Score feature



Visualize label separation with upvote ratio feature

Looking at the graphs, it seems like -

- in the case of the comments feature, artificial is easily separable by the other two classes
- in the case of the upvote_ratio feature, it does not seem to separate data as clearly
- in the case of the score feature, the artificial label is generally lower, hence relatively easier to separate

Let's verify these hypotheses as we tune the random forest model

```
## prepare datasets
X = data[~data.flair_text.isna()]
X['flair_text'] = data.flair_text.dropna().astype(str).apply(preprocess)
y, y_org = data[~data.flair_text.isna()].subreddit.factorize()

## split the dataset into train-test-valid stratified split; to keep the ratio o
X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    test_size=0.15,
                                                    random_state=42,
                                                    stratify=y)

X_train, X_val, y_train, y_val = train_test_split(X_train,
                                                  y_train,
                                                  test_size=0.10,
                                                  random_state=42,
                                                  stratify=y_train)
X_train.shape, y_train.shape
```

Since we are using categorical and numerical features, we need to do some preprocessing.

- We need to one-hot-encode the categorical features.

- We should normalize the numerical features using scaling.

```
## Define the preprocessor
## Use One-hot-encoding to process the categorical (textual) variables
## Use scaling to normalize numeric features
preprocessor = ColumnTransformer([('cat', OneHotEncoder(), ['flair_text']),
                                  ('num', StandardScaler(), ['score',
                                                             'comments',
                                                             'upvote_ratio',
                                                             'year'])])

# Define the pipeline
clf = Pipeline([('preprocessor', preprocessor), ('clf', RandomForestClassifier(n
                                                                                r

## fit the pipeline
clf.fit(X_train, y_train)

## model performance
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)

## removing one of the data points because it is not present in one-hot-encoded
X_val, y_val = X_val[~X_val.flair_text.isin(['opinion'])], y_val[~X_val.flair_te
y_pred_val = clf.predict(X_val)

print(f'Train Accuracy - {round(100*accuracy_score(y_train, y_pred_train),2)}%')
print(f'Test Accuracy - {round(100*accuracy_score(y_test, y_pred_test),2)}%')
print(f'Validation Accuracy - {round(100*accuracy_score(y_val, y_pred_val),2)}%\

print(f'Classification Report - \n{classification_report(y_test, y_pred_test)}')
```

Output —

```
Train Accuracy - 100.0%
Test Accuracy - 91.26%
Validation Accuracy - 85.92%

Classification Report -
              precision    recall  f1-score   support

           0       0.87      0.95      0.91       138
           1       0.91      0.89      0.90       130
           2       0.99      0.89      0.94        98

    accuracy                           0.91       366
   macro avg       0.92      0.91      0.91       366
weighted avg       0.92      0.91      0.91       366
```

Observations

- The overall model performance is good, above 90%
- However, there is some overfitting
- The model is still useful as it is ~90% accurate for testing and validation datasets

Let's confirm our hypothesis by looking at the feature importance assigned by the random forest classifier.

```
# Get the feature importances
importances = clf.named_steps['clf'].feature_importances_

# Create a dataframe with feature names and importances
feature_importances = pd.DataFrame({'Feature': ['flair_text_'+str(i) for i in ra
                                    'Importance': importances})

# Sort the features by importance in descending order
feature_importances = feature_importances.sort_values(by='Importance', ascending

# Print the feature importances
print(feature_importances)
```

Output -

```
          Feature  Importance
24          score    0.380288
25       comments    0.200198
27           year    0.068081
26   upvote_ratio    0.055558
18  flair_text_18    0.048130
0    flair_text_0    0.047541
2    flair_text_2    0.041683
14  flair_text_14    0.037163
15  flair_text_15    0.033372
5    flair_text_5    0.020604
3    flair_text_3    0.020573
9    flair_text_9    0.015279
22  flair_text_22    0.007973
11  flair_text_11    0.005218
16  flair_text_16    0.004599
20  flair_text_20    0.004339
1    flair_text_1    0.001462
21  flair_text_21    0.001351
4    flair_text_4    0.001214
6    flair_text_6    0.001080
13  flair_text_13    0.000952
7    flair_text_7    0.000768
10  flair_text_10    0.000544
12  flair_text_12    0.000502
```

```
19  flair_text_19    0.000467
8    flair_text_8    0.000461
23  flair_text_23    0.000422
17  flair_text_17    0.000176
```

Even if we remove the categorical feature and only work with numerical features, we get the following output.

```
Train Accuracy - 85.24%
Test Accuracy - 76.78%
Validation Accuracy - 73.79%

Classification Report -
              precision    recall  f1-score   support

           0       0.70      0.84      0.76       138
           1       0.73      0.62      0.67       130
           2       0.94      0.86      0.90        98

    accuracy                           0.77       366
   macro avg       0.79      0.77      0.78       366
weighted avg       0.78      0.77      0.77       366
```

Observations -

- Even without any textual data, the model is able to capture some trends and be significantly accurate.
- ~75% accuracy for multiclass problems is decent.
- The model seems to be less overfitting as well and pretty consistent in the case of testing and validation dataset
- As discussed earlier, the most important feature turned out to be the score and number of comments. The year turned out to be less important for the random forest classifier with this configuration.

Support Vector Machines

Support Vector Machine (SVM) is a powerful and widely used machine learning algorithm for classification, regression, and outlier detection. It works by finding the hyperplane that separates the data points in a high-dimensional feature space with a maximum margin, where the margin is defined as the distance between the hyperplane and the closest data points. SVM can handle both linear and non-linear datasets using different kernel functions that map the data into a higher-dimensional space. SVM is known for its ability to generalize well and perform well on high-dimensional datasets.

For this specific dataset, SVM can be a good choice because it is a relatively small dataset with a moderate number of features. SVM can handle both categorical and numerical features, which are present in this dataset and can be used for multi-class classification, which is the task for this dataset. However, SVM's performance may be affected by the imbalance in the target variable classes, as we saw in the Random Forest Classifier example.

> ***Experiment 3. A — flair_text and Non-Textual features***

```
## Define the preprocessor
preprocessor = ColumnTransformer([('cat', OneHotEncoder(), ['flair_text']),
                                  ('num', StandardScaler(), ['score','comments',

## Define the pipeline
clf = Pipeline([('preprocessor', preprocessor), ('clf', SVC(kernel='linear', C=1

## fit the model
clf.fit(X_train, y_train)


## predict
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)
y_pred_val = clf.predict(X_val)

print(f'Train Accuracy - {round(100*accuracy_score(y_train, y_pred_train),2)}%')
print(f'Test Accuracy - {round(100*accuracy_score(y_test, y_pred_test),2)}%')
print(f'Validation Accuracy - {round(100*accuracy_score(y_val, y_pred_val),2)}%\

print(f'Classification Report - \n{classification_report(y_test, y_pred_test)}')
```

Output -

```
Train Accuracy - 87.49%
Test Accuracy - 87.7%
Validation Accuracy - 84.47%

Classification Report -
              precision    recall  f1-score   support

           0       0.85      0.89      0.87       138
           1       0.84      0.86      0.85       130
           2       0.98      0.88      0.92        98

    accuracy                           0.88       366
   macro avg       0.89      0.88      0.88       366
weighted avg       0.88      0.88      0.88       366
```

Observations

- The SVM algorithm also performs really well on this dataset.
- The model performance is highly balanced and is in a great ballpark as well.
- Both the precision and recall are high, which is a good sign for a balanced model.

Hyperparameter Tuning

```
# Define the preprocessor
preprocessor = ColumnTransformer([('cat', OneHotEncoder(), ['flair_text']),
                                  ('num', StandardScaler(), ['score','comments',

# Define the pipeline
clf = Pipeline([('preprocessor', preprocessor), ('clf', SVC(kernel='rbf', C=3))]

## fit the classifier
clf.fit(X_train, y_train)

## predict
y_pred_train = clf.predict(X_train)
y_pred_test = clf.predict(X_test)
y_pred_val = clf.predict(X_val)

print(f'Train Accuracy - {round(100*accuracy_score(y_train, y_pred_train),2)}%')
print(f'Test Accuracy - {round(100*accuracy_score(y_test, y_pred_test),2)}%')
print(f'Validation Accuracy - {round(100*accuracy_score(y_val, y_pred_val),2)}%\

print(f'Classification Report - \n{classification_report(y_test, y_pred_test)}')
```

Output —

```
Train Accuracy - 90.23%
Test Accuracy - 88.25%
Validation Accuracy - 85.44%

Classification Report -
              precision    recall  f1-score   support
```

```
           0       0.85      0.89      0.87       138
           1       0.88      0.88      0.88       130
           2       0.93      0.88      0.91        98

    accuracy                           0.88       366
   macro avg       0.89      0.88      0.88       366
weighted avg       0.88      0.88      0.88       366
```

Observations

- After hyper-parameter tuning, the accuracy increases slightly
- The best hyperparameters are — Kernal: 'rbf', C : 3
- Just like before, the model's performance is highly balanced
- The model is >85% accurate for all testing and validation datasets
- The model does not show overfitting as well, which can be seen by comparing train, test, and validation accuracies

Performance Comparison

As I said earlier, we have been saving all the models, their configuration, and their output to compare later. This comparison will tell us which models are the best and how better are they than others.

Let us compare the performances of all the models that we trained above.

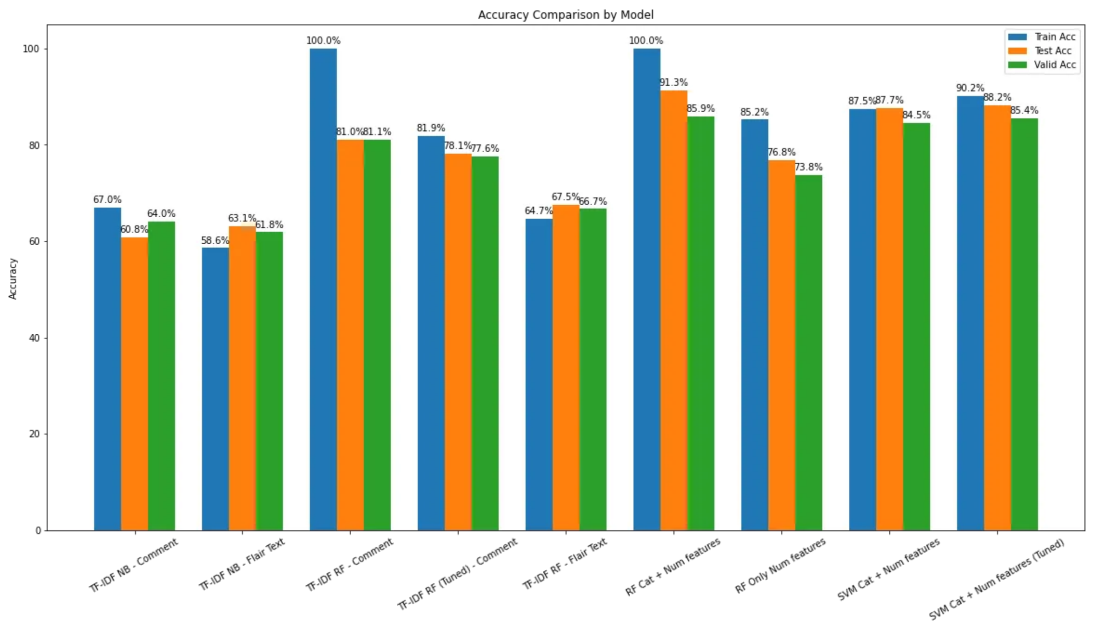
The perf_comp dictionary looks something like following after capturing the information of all the experiments -

```
{'title': ['TF-IDF NB - Comment',
  'TF-IDF NB - Flair Text',
  'TF-IDF RF - Comment',
  'TF-IDF RF (Tuned) - Comment',
  'TF-IDF RF - Flair Text',
  'RF Cat + Num features',
  'RF Only Num features',
  'SVM Cat + Num features',
  'SVM Cat + Num features (Tuned)'],
 'train_acc': [66.96, 58.56, 100.0, 81.89, 64.68, 100.0, 85.24, 87.49, 90.23],
 'test_acc': [60.8, 63.11, 81.03, 78.12, 67.49, 91.26, 76.78, 87.7, 88.25],
 'valid_acc': [64.02, 61.84, 81.1, 77.56, 66.67, 85.92, 73.79, 84.47, 85.44],
 'clf': [Pipeline(steps=[('tfidf', TfidfVectorizer()), ('clf', MultinomialNB())]
  Pipeline(steps=[('tfidf', TfidfVectorizer()), ('clf', MultinomialNB())]),
  Pipeline(steps=[('tfidf', TfidfVectorizer()),
                  ('clf', RandomForestClassifier(random_state=42))]),
  Pipeline(steps=[('tfidf', TfidfVectorizer()),
                  ('clf',
                   RandomForestClassifier(max_depth=10, min_samples_leaf=10,
                                          min_samples_split=50,
                                          random_state=42))]),
  Pipeline(steps=[('tfidf', TfidfVectorizer()),
                  ('clf',
                   RandomForestClassifier(n_estimators=50, random_state=42))]),
  Pipeline(steps=[('preprocessor',
                   ColumnTransformer(transformers=[('cat', OneHotEncoder(),
                                                    ['flair_text']),
                                                   ('num', StandardScaler(),
                                                    ['score', 'comments',
                                                     'upvote_ratio', 'year'])]))
                  ('clf', RandomForestClassifier(random_state=42))]),
  Pipeline(steps=[('preprocessor',
                   ColumnTransformer(transformers=[('num', StandardScaler(),
                                                    ['score', 'comments',
                                                     'upvote_ratio', 'year'])]))
                  ('clf',
                   RandomForestClassifier(max_depth=10, min_samples_leaf=5,
                                          n_estimators=50, random_state=42))]),
  Pipeline(steps=[('preprocessor',
                   ColumnTransformer(transformers=[('cat', OneHotEncoder(),
                                                    ['flair_text']),
                                                   ('num', StandardScaler(),
                                                    ['score', 'comments',
                                                     'upvote_ratio', 'year'])]))
                  ('clf', SVC(C=1, kernel='linear'))]),
  Pipeline(steps=[('preprocessor',
                   ColumnTransformer(transformers=[('cat', OneHotEncoder(),
                                                    ['flair_text']),
                                                   ('num', StandardScaler(),
                                                    ['score', 'comments',
                                                     'upvote_ratio', 'year'])]))
                  ('clf', SVC(C=3))])]}
```

We can interpret this information using the matplotlib library.

```
# set the x-axis labels
labels = perf_comp['title']

# set the y-axis data
train_data = perf_comp['train_acc']
test_data = perf_comp['test_acc']
valid_data = perf_comp['valid_acc']

# create the bar chart
x = np.arange(len(labels))
width = 0.25

fig, ax = plt.subplots(figsize=(20,10))
train_bars = ax.bar(x - width, train_data, width, label='Train Acc')
test_bars = ax.bar(x, test_data, width, label='Test Acc')
valid_bars = ax.bar(x + width, valid_data, width, label='Valid Acc')
```

```
# add labels, title, and legend
ax.set_ylabel('Accuracy')
ax.set_title('Accuracy Comparison by Model')
ax.set_xticks(x)
ax.set_xticklabels(labels, rotation=30)
ax.legend()

# add values above each bar
def label_plot(bars):
    for bar in bars:
        height = bar.get_height()
        ax.annotate(f'{height:.1f}%',
                    xy=(bar.get_x() + bar.get_width() / 2, height),
                    xytext=(0, 3),
                    textcoords='offset points',
                    ha='center', va='bottom', )

label_plot(train_bars)
label_plot(test_bars)
label_plot(valid_bars)

plt.show()
```



Compare the accuracies of each model

Contribution

The goal was to build a classifier algorithm to predict the categorical label — subreddit. The output has to be one of the classes — MachineLearning, datascience, or artificial. I have tried to compare multiple models and their hyper-parameters tuned versions to compare the model performances. This will tell us which model is the best for this particular problem and its best configuration as well.

- I have worked with 4 different kinds of models: TF-IDF Naive Bayes, TF-IDF Random Forest Classifier, Random Forest Classifier, and Support Vector Machine.
- In total, we have 9 model configurations to compare.
- I have tuned hyper-parameters wherever needed and see the output.
- I have stored every model configuration and output into a dictionary to compare all the models at the end of this exercise.
- In the plot above, we can observe the best model, and its configuration is saved as an instance in the perf_comp dictionary as well.
- In the total 9 experiments, I have worked with both categorical and numerical features either together or alternatively to find out which features provide the best prediction of the target label.
- The best model comes out to be SVM with both categorical and numerical features. We can see that the model accuracy is the best among others without overfitting issues. The accuracy of 85% in the validation dataset and 88% in the testing dataset is a great indicator for a multiclass model.
- In the classification reports mentioned with each experiment, we can analyze that sometimes even if the accuracy is substantial, the model is not good for real-world use because of the worst performance in one of the classes. This was the case with experiment 1A, where we are building a Multinomial Naive Bayes classifier using TF-IDF vectorization.
- On the other end, the SVM model has the most balanced classification report.
- Hyperparameters are tuned for all the overfitting cases, where the testing and validation accuracy is substantial to analyze the real-world performance of the respective model.

References

I have referred to some articles and documentation to understand the concepts used in this Project. Links are mentioned below -

- https://nlp.stanford.edu/IR-book/html/htmledition/support-vector-machines-and-machine-learning-on-documents-1.html#:~:text=An%20SVM%20is%20a%20kind,points%20as%20outliers%20or%20noise).
- https://medium.com/analytics-vidhya/nlp-tutorial-for-text-classification-in-python-8f19cd17b49e
- https://www.analyticsvidhya.com/blog/2018/04/a-comprehensive-guide-to-understand-and-implement-text-classification-in-python/
- https://medium.com/@tenzin_ngodup/simple-text-classification-using-random-forest-fe230be1e857

Text Classification   Reddit Post Classifier