The first step of people finding jobs is looking for job postings, either online or offline. With the emergence of the Internet and several platforms (e.g. Scouted, Jora, Indeed), job seekers have more access to free, available job postings. However, this convenience meanwhile increases the risk of encountering fraudulent postings, harming privacy and security.

This article explains how to build and optimize a model that is accurate and validated in not only the used dataset but also some new estimators and how to evaluate the model's performance on recognizing real vs. fake job postings. The dataset used and the code implemented are provided in link forms at the end of this article.

Once the dataset is loaded into a data frame, explorations are performed to understand its data types, columns, shapes, and other information. Some of the features, like job_id, are not needed to do the prediction while some contain a lot of null values, so deleting unnecessary columns and handling missing values are vital steps prior to model building.

```
Data columns (total 13 columns):
 #   Column               Non-Null Count  Dtype
---  ------               --------------  -----
 0   title                1917 non-null   object
 1   location             1917 non-null   object
 2   salary_range         1917 non-null   object
 3   requirements         1917 non-null   object
 4   telecommuting        1917 non-null   int64
 5   has_company_logo     1917 non-null   int64
 6   has_questions        1917 non-null   int64
 7   employment_type      1917 non-null   object
 8   required_experience  1917 non-null   object
 9   required_education   1917 non-null   object
 10  industry             1917 non-null   object
 11  function             1917 non-null   object
 12  fraudulent           1917 non-null   int64
dtypes: int64(4), object(9)
memory usage: 209.7+ KB
```

Figure 1 Data frame information after pre-processing

Above are examined features. It is clear that 'fraudulent' is the target variable, while the rest make up the feature variable.

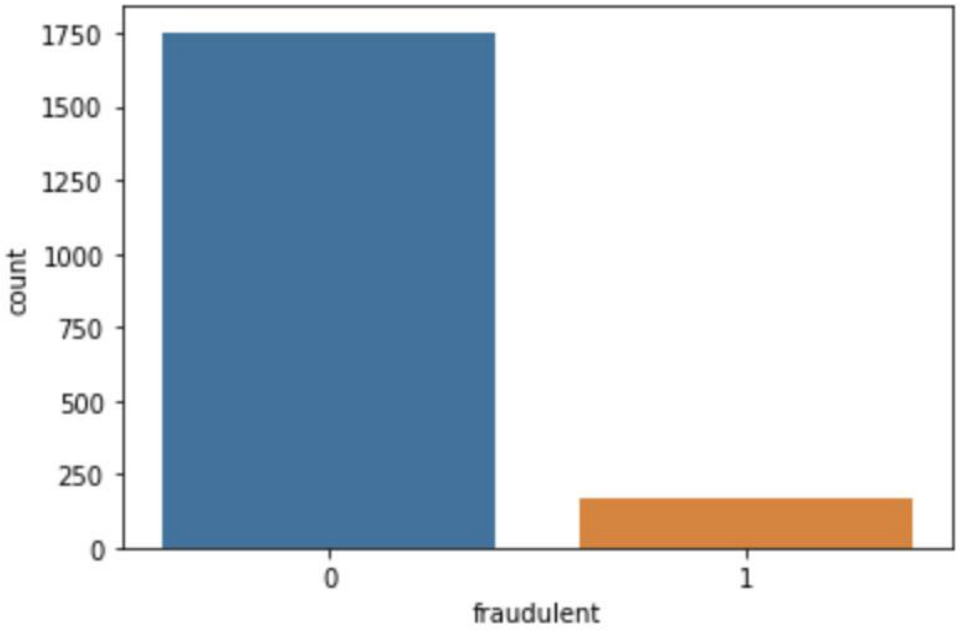

Figure 2 Count plot of target variable: ‘0’ for not fraudulent job posting and ‘1’ for the fraudulent job posting

We can observe that the target variable is unbalanced, so when partitioning data, stratify and random state argument are preferred. After having the train/test sets ready, I experiment with different ensembles/models to fit the training data and predict the testing ones. Experimented are DummyClassifier(), LogisticRegression(), BaggingClassifier(), RandomForestClassifier(), AdaBoostClassifier(), GradientBoostingClassifier(), VotingClassifier(), KNeighborsClassifier(), and XGBClassifier(), generating the following results.

	model	accuracy
0	baseline	0.914062
1	logistic regression	0.953125
2	bagging	0.942708
3	random forest	0.971354
4	ada boost	0.973958
5	gradient boosted trees	0.971354
6	voting	0.971354
7	k nearest neighbors	0.973958
8	xg boost	0.976562

Figure 3 Model names and their corresponding accuracy in predicting job postings

Extreme gradient boosting (XGBoost) is chosen as the model to be further elaborated on, as it has the highest accuracy and contains many hyperparameters to be tuned and choices for regularization to perform a more accurate while not under-/over-fitting model.

I first set several hyperparameters to their typical values. Then, I perform cross-validation to see what score the current model would get. The min of test RMSE means is 0.188636. The reason why cross-validation is needed for tuning hyperparameters (instead of just looking at the accuracy) is that the model that fits this set might not be able to apply to other estimators.

When trying approaches to tune hyperparameters, I find that both for loops of cross-validation and grid/randomized search take an extremely long time (above an hour). So, I use cross-validation by manually inputting hyperparameters to try out the best fit.

I find the following is the best combination (least RMSE) for this model within a certain range: n_estimators=100, max_depth 7, learning_rate=0.25, and subsample=0.9. Though subsample reduces the model's RMSE, it is still needed for pruning to reduce the chances of over-fitting.

After implementing these parameters, we can alter other ones to avoid overfitting. We've already included learning_rate, aka eta, in regularization (multiplying the tree values by a number less than one to make the model fit slower and prevent overfitting) and subsample in sampling (subsample rows of the training data prior to fitting a new estimator).

We can do more of these to avoid overfitting. Next, I would perform the search (with some predetermined hyperparameters) to find the optimal min_split_loss, aka gamma, in pruning (fixed threshold of gain improvement to keep a split), L2 regularization, aka lambda, and L1 regularization, aka alpha (a shift which splits are taken and shrink the weights). XGBoost itself has these parameters to regulate the model. An increase in alpha/lambda means a more conservative model.

When figuring out the optimal gamma, I employ GridSearchCV(). This method is computationally expensive and a bit time-consuming. RandomizedSearchCV() can do a similar work yet in a much faster way.

With all optimized hyperparameters being identified, I implement them into the XGBClassifier() to revise the original one into a better version. The following classification report and testing data confusion matrix are generated. The model's accuracy increases to 98%, while it also takes regularization into concern.

Classification Report:

	precision	recall	f1-score	support
0	0.98	0.99	0.99	351
1	0.93	0.79	0.85	33
accuracy			0.98	384
macro avg	0.95	0.89	0.92	384
weighted avg	0.98	0.98	0.98	384

Figure 4 Classification report of the optimized XGBoost model containing precision, recall, f1-score, and support

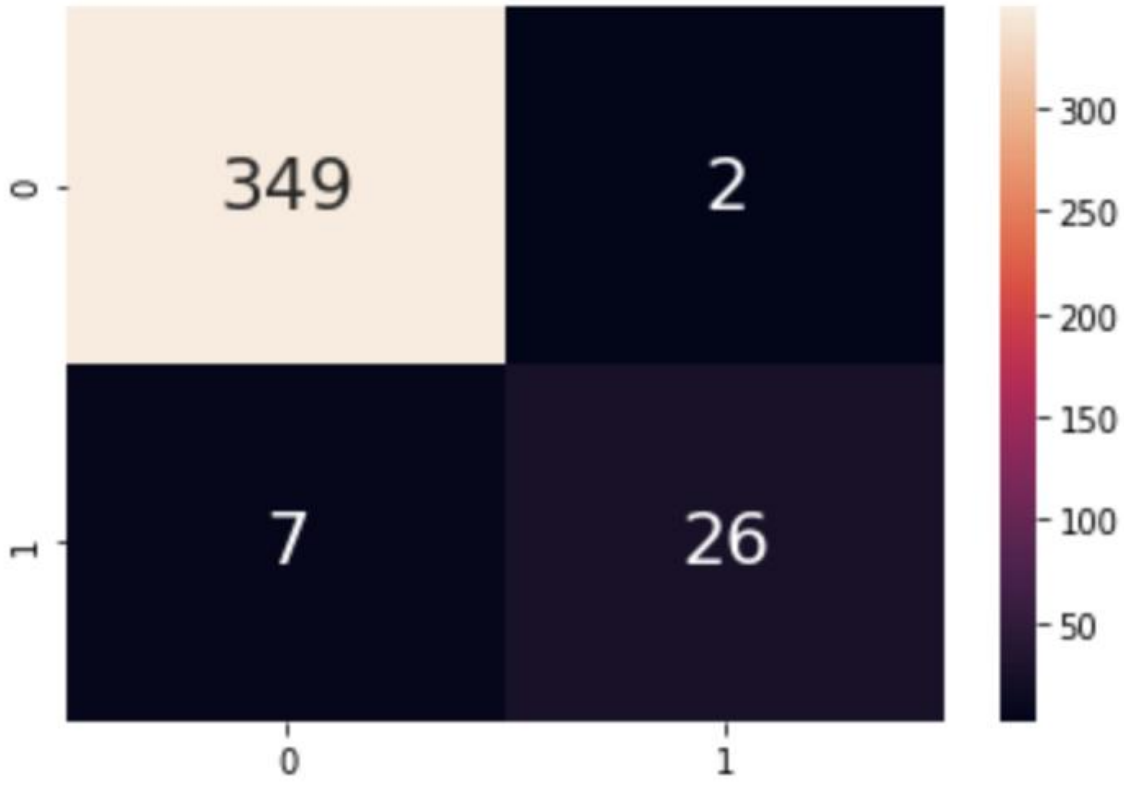

Figure 5 Confusion matrix for testing data

After the model is revised, I perform another cross-validation. Due to the imbalance in instances for each class (recall the count plot—far more 'not fraudulent than 'fraudulent,' this time, I conduct stratified k-fold cross-validation to evaluate the optimized XGBoost model and get a score of 95.50% (0.84%).

As many features examined are categorical variables, I change them into dummy variables when establishing the model. I perform a simple for loop to integrate feature importance into the original columns, arriving at the graph below.

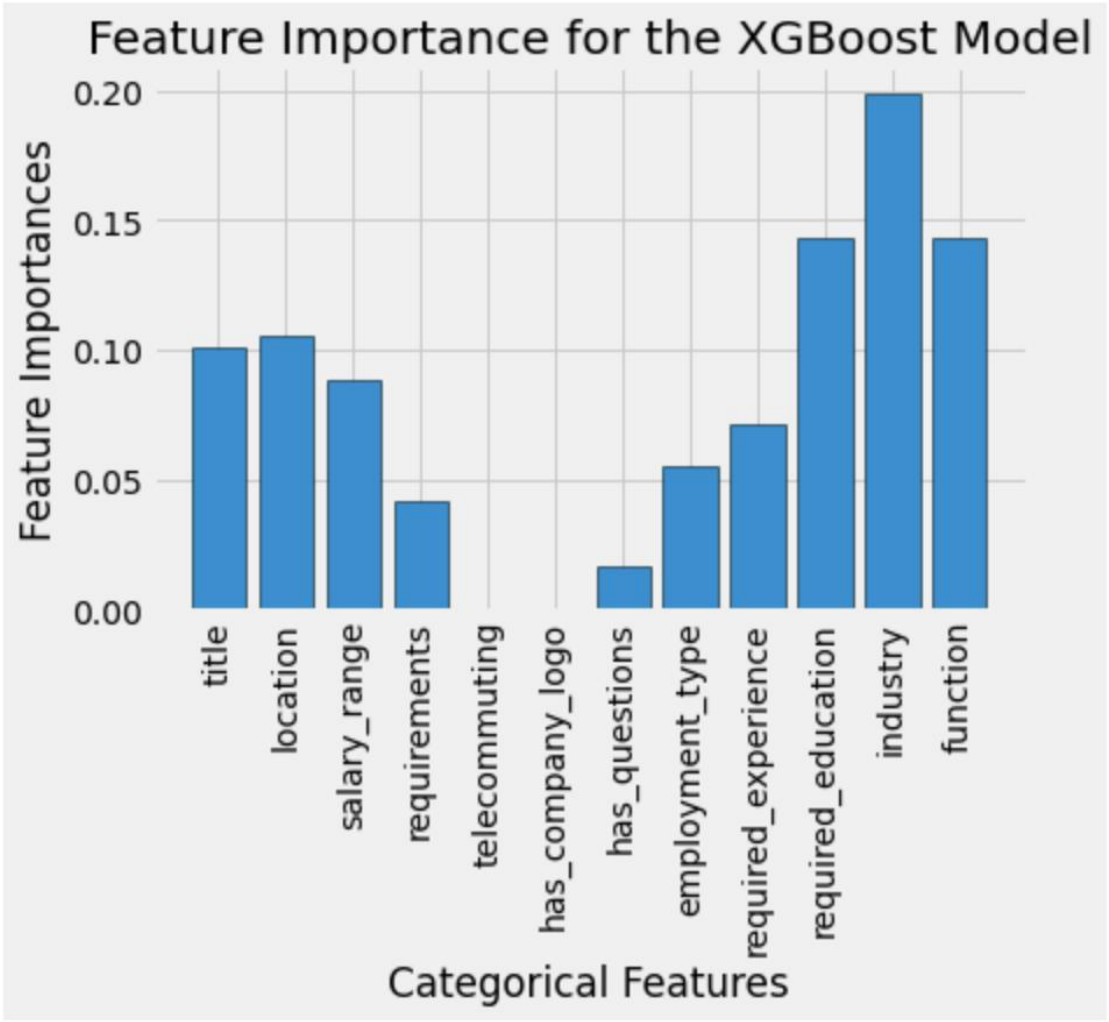

Figure 6 Feature importance of the categorical features for the XGBoost model

The industry category variable has the most feature importance, meaning that the kind of industry matters the most when deciding the authenticity of job postings. Thus, I explore more on this.

I generate a graph of several industries with the top percentage of fraudulent job postings. The percentage here is not the ratio of fraudulent job postings in the industry over the total fraudulent, as the dataset contains unequal amounts of information about each industry. The percentage is the ratio of the industry's fraudulent postings over the industry's total job postings in the dataset. Otherwise, the percentage would be biased and underrepresented. Also, I set the industry's total posting as at least 10 to

be in the graph (a small sample size can't really tell a story—having just 1 job posting in a particular industry and is fake generates a 100% fraudulent percentage).

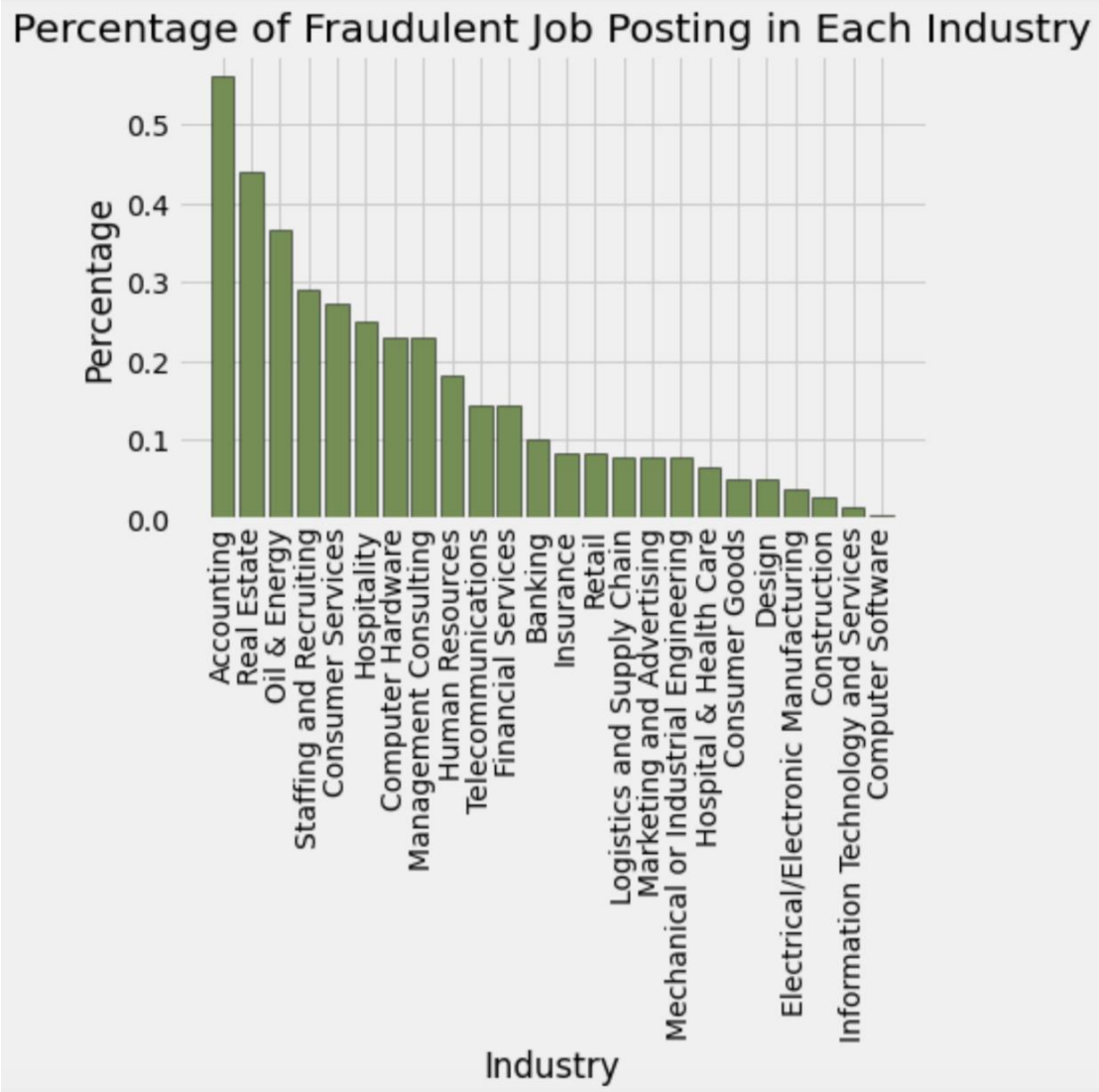

Figure 7 Percentage of Fraudulent Job Posting in Each Industry (according to the industry's category data set size)

We can conclude that people seeking jobs in accounting, real estate, and oil&energy should be especially careful. With the skills of modeling in machine learning, we can even minimize the opportunities of leaking personal and contact information!

Data set link:
https://www.kaggle.com/shivamb/real-or-fake-fake-jobposting-prediction