Wine Quality Regression Problem

Gabriele Rosi
Politecnico di Torino

***Abstract*—The main objective of this project is to predict the wine quality given some information about the geographic location, the winery, the grapes variety and a general description. Starting from a preliminary data analysis, we apply some preprocessing techniques on the available data such as tokenization/lemmatization, TF-IDF and One-Hot Encoding. Then we train a Ridge regression model to predict the wine quality.**

I. Problem overview

The proposed task is a regression problem on a dataset containing information about wines. The goal is to correctly predict the wine quality. The dataset provided to achieve this task is divided into two file: `dev.tsv` and `eval.tsv`. Both files are simple text file were each value is separated by a tab character (indeed the name acronym ”tsv” stands for tab separated values). The only difference (in term of columns) between the two files is about the **quality** column: this is the target value that we need to predict. This column is available in `dev.tsv` but not in `eval.tsv`. Therefore the data contained in `dev.tsv` can be considered as the *”Development set”* and the data contained in `eval.tsv` as the *”Evaluation set”*. In the development set we have 120744 records and 9 columns. Instead in the evaluation set we have 30186 records and 8 columns. Here a summary about the columns available in the two set:

- **country**, contains the country were the wine is produced;
- **description**, contains a brief description about the wine;
- **designation**, contains information about wine designation (e.g. reserve, estate, barrel sample...);
- **province**, **region_1**, **region_2** contains geographic information;
- **variety**, contains information about wine variety (e.g. red blend, Sangiovese, Chardonnay...);
- **vinery**, contains information about the winery that produced the wine;
- **quality** *(available only for* `dev.tsv`*)*, quality of the wine.

Now we are going to analyze more in deep the development set. Such a big number of records, suggests to make an analysis starting from duplicates data. As we expect, we have 35716 duplicates value. We choose to drop them in order to achieve a better result, since duplicates data can lead our model to overfitting. Moreover if a row has got 1000 duplicates, we can have an unbalanced problem. So, from now on, we are going to consider the development set without any duplicates. For this reason in the development set now we have 85028 records with the same number of columns (9).

Analyzing every single columns we can also observe that there are a lot of missing values. In table I there is a summary report. As we can see they are almost all in three columns. Since the majority of regression algorithms can not handle these, we are going to fill them in some way.

Column	**Attribute type**	**Missing values**	**% of missing values**
country	*categorical*	3	$\sim$ 0%
description	*categorical*	0	0%
designation	*categorical*	25944	30.51%
province	*categorical*	3	$\sim$ 0%
region_1	*categorical*	13889	16.34%
region_2	*categorical*	50734	59.67%
variety	*categorical*	0	0%
winery	*categorical*	0	0%

TABLE I: Column summary, with overview of missing values.

As we could expect the quality column does not have any missing values. The values are bound in a range of values [0, 100] with μ = 46.42 and σ = 11.93. In figure 1 we can see the distribution of data.

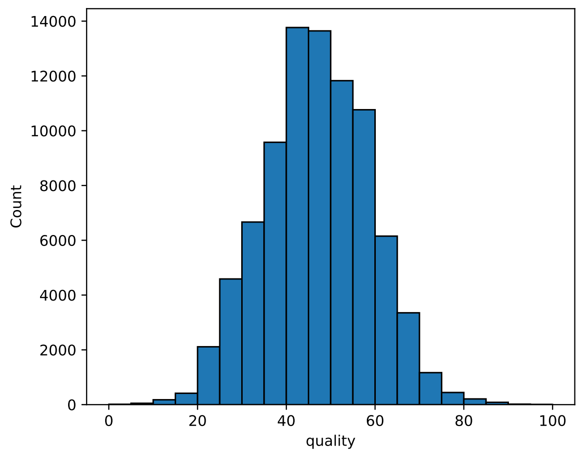

Fig. 1. Distribution of wine quality in development set.

II. Proposed approach

Let’s focus now on the proposed approach: we are trying to explain the steps to achieve the final result, in order to make this process reproducible.

A. *Preprocessing*

As we already seen in table I, all the attributes that we have are categorical. This means that are textual data and we need to transform them in order to make ”understandable” for our model.

As we said before, first thing to do is **drop duplicates values**, because can cause many problems to our model.

Also we **fill all our missing values** with a dummy character, in particular an empty character (””).

Another preliminary step to do is to **lowercase** all our data. Since they are all textual data, *”word”* is different from *”Word”* or *”WORD”*, for example. From this, some kind of transformations will generate three different values without lowercase, instead of only one with lowercase.

Let’s go more in deep with transformation for each column. We leave for a moment *description* and we concentrate on the other columns. These columns contains only some textual indication that are independent each other. For these reason we have choosen to transform them using **One-Hot Encoding** (OHE). One-Hot Encoding is a common technique used in machine learning pipelines to transform textual data into machine readable information. Such transformation consist of encoding each value in a vector of *M* element, where *M* is the number of distinct value for the attribute. Then in each vector we have only one element with value = 1 and the others = 0. In this way we are creating a numerical vector that encodes each unique value of our categorical attributes in a unique way (e.g. if we have `[[Red], [Green], [Blue]]`, then the One-Hot encoded version is: `[[0,0,1], [0,1,0], [1,0,0]]`). This process makes these information machine readable. But we have also another benefit: one can think to encode them in such a way that each value correspond to an incremental integer value in range [0, M], where *M* is the number of distinct value for the attribute. But in this way we are saying to our model that $X_1 < X_2 < ... < X_M$ and this is not a correct behavior since our values cannot be ordered. This kind of transformation has been applied to *country*, *designation*, *province*, *region_1*, *variety*, *winery*. In table II we can see a summary of the number of unique values identified and consequently the *M* dimension of each OHE vector. Notice that *country*, *designation*, *province*, *region_1* contains one more element in the OHE vector because these four columns contains missing values (represented as an empty character) that OHE encodes in a unique value.

Column	**Unique values**	***M (dimension of OHE vector)***
country	48	49
designation	27635	27636
province	444	445
region_1	1206	1207
variety	603	603
winery	14102	14102
TOTAL		44042

TABLE II: Summary of OHE transformation.

Now we are going to analyze *description*. As mentioned before, *description* contains a general description about the wine. This means that all the description has some information in common (e.g. a word can appear in more than one description and we need to take care of this occurrences). So to take into account also the frequency of every single word, we have choosen to transform this columns using **TF-IDF (Terms Frequency - Inverse Document Frequency)**. This techniques helps to reflect how important a word is to a document in a collection. Given a term t in document d of collection D consisting of m documents, TF-IDF is computed as follows: $tfidf(t) = freq(t, d) * log\left(\frac{m}{freq(t,D)}\right)$. This means that terms occurring frequently in a single document but rarely in the whole collection are preferred. This can help us to characterize each single wine and leave out of the scope such words that are very often and common between wines. To help TF-IDF works better we introduce some particular preprocessing steps on *description*. In particular we created a custom tokenizer process used by the TF-IDF function. When we talk about tokenizing, we are referring to the process that splits a phrase (or a speech) into chunks by dividing words using the whitespace as separator. Therefore, given a document, the main operation to do before TF-IDF are:

1) Remove from the document every character that is not a letter (a-z or A-Z) or a whitespace.
2) Then tokenize the document and for each token:
    a) Remove any blank space at the beginning or at the end.
    b) Lemmatize the token.
    c) Discard the token if $length < 3$.

With lematization we are trying to determine the lemma of a word based on its meaning. For this reason word like *run*, *running*, *ran* are converted on the same lemma ”run”. With this technique we are able to reduce the number of word that we are going to process with the TF-IDF. During the TD-IDF process, we are removing also all the English stop-words using the list provided by the NLTK library [1]. The TF-IDF transformation produce 37848 new features.

Since all the transformation result’s are rich of zero values (especially for the OHE vectors), we decided to store the result matrix, containing the OHE vectors and the TF-IDF vectors, into a **sparse matrix**. If now we sum the features produced by OHE and the TF-IDF we obtain a matrix with 85028 rows and 81890 columns. With sparse matrix we have ”only” 2403519 values, instead of 6962942920 without!

For the preprocessing step we ignore *region_2* because it has got too many missing value, as you can see in table I.

B. Model selection

The following algorithms has been selected also based on the help provided by the scikit-learn algorithm cheat sheet [2]:

- *SVR*: linear model that gives some flexibility about how much error is acceptable for our model. As we can see in the algorithm cheat sheet, is advised to use this algorithm when we have $\#samples < 100$ and almost all the feature are important;
- *Lasso*: linear model that allows to reduce the model complexity, prevent overfitting and help in feature selection since it tends to assign values very close to zero to

some coefficient. As we can see in the algorithm cheat sheet, is advised to use this algorithm when we have $\#samples < 100$ and only some feature are important;

- *Ridge*: linear model that allows to reduce the model complexity and prevent overfitting. Ridge help also in case of multi-collinearity: this occurs when two or more independent variables in the dataset are correlated with each other. OHE is affected by this problem [3] and this algorithm can help to resolve this. Ridge in contrast to Lasso tends to lower uniformly all the coefficients;
- *Random forest*: this algorithm build different decision trees that are trained on different data sample with replacement. When all the trees are builded it combines together to determine the final output. This technique usually control overfitting. We chose this model since it work great with large data and for it's capability to avoid overfitting.

C. Hyperparameters tuning

Since we have 4 model to test, we have lot of hyperparameters available. For this reason we decided to run each algorithm on our data with the default parameters in order to exclude a priori some of them. We can use an 80/20 train/test split on the development set and measure the R2 score. In the table III we can see the results.

Model	**Results**	***Conclusion***
SVR	After one hour of training it has not finished yet. Probably our dataset has got to much features to take into account.	Excluded
Lasso	Very quick train but strange and negative R2 score	Excluded
Ridge	Very quick train and very good R2 score	**Potentially a good choice**
RandomForest	Long train but good R2 score	**Potentially a good choice**

TABLE III: Algorithms train with default parameters.

We can see that the remaining models after this little experiment are: *RandomForest* and *Ridge*. Now it's more easy to explore the hyperparameters space since their number is smaller. We can now run a gridsearch on both Ridge and RandomForest, based on the hyperparameters in table IV.

Model	**Hyperparameters**	**Values**
Ridge	*alpha*	{0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1.0}
	fit_intercept	{True, False}
RandomForest	*n_estimators*	{100, 200}
	min_sample_split	{2, 10, 20}

TABLE IV: Hyperparameters considered in gridsearch exploration.

III. Results

After the hyperparameters tuning with a cross validation with five fold, we have found the two best configuration for our models are the following:

- **RandomForest**: {*n_estimators = 200, min_sample_split = 2*} with an R2 score = 0.5547
- **Ridge**: {*alpha = 0.8, fit_intercept = True*} with an R2 score = 0.7493

Ridge is way more better than RandomForest. This may be because of the fact Ridge can "resolve" the problem of multicollinearity.

In figure 2 we can see a comparison between the mean[1] fit time and the mean R2 score for each hyperparameter configuration for Ridge model. Low value of alpha (0.1 or 0.2) got a higher mean fit time with smaller mean R2 score. Instead, values of mean fit time and mean R2 score that are near to our best model[2], are very similar and they are dependent to *fit_intercept* value (*True or False*).

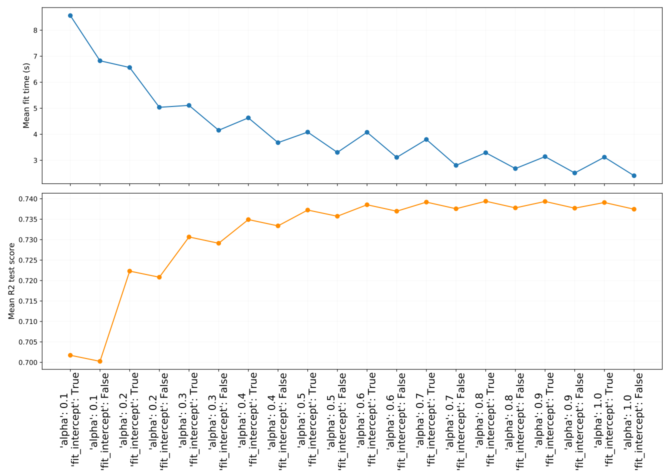

Fig. 2. Comparison between mean fit time and mean R2 score for each hyperparameter configuration for Ridge model.

After choosing Ridge as the final model, we trained it on the whole development set and we test it on the evaluation set. The public score obtained is 0.830 that is a good score. The model seems to generalize well the problem and it can be reasonable to think that there is no overfitting.

IV. Discussion

The proposed approach brings a good result but it can be improved as well. Probably we can try to extract some features by analyzing the distribution of the wine quality and the geographical location. Another possible improvement can be to better transform the *description* columns with some techniques that can obtains more accurate results also based on the document content (like sentiment analysis).

References

[1] NLTK, "Nltk documentation homepage." https://www.nltk.org/index.html.

[2] scikit learn, "scikit-learn algorithm cheat sheet." https://scikit-learn.org/stable/tutorial/machine_learning_map/index.html.

[3] K. K. Mahto, "One-hot-encoding, multicollinearity and the dummy variable trap." https://towardsdatascience.com/one-hot-encoding-multicollinearity-and-the-dummy-variable-trap-b5840be3c41a.

[1] mean because we are doing cross validation

[2] {*alpha = 0.8, fit_intercept = True*}