IMT 574 Problem Set 6

Deadline: Fri, Mar 15th midnight

Introduction

This problem set focuses on attribute selection, regularization, and cross-validation (CV). We use rotten tomatoes data, a dataset about text which leads to very bad design matrices (think about sparsity and multicollinearity). We do our best to predict whether the tomato is fresh or rotten, based on this data. However, while the rationaly behind the evaluation is described in the whole review, we only observe a single sentence (quote) from it. So don't expect too good results! You see that several shiny methods we learned are not working at all, while other do reasonably well, and are not at all disturbed by the size and sparsity of it. Some of the models may be awfully slow requiring you to wait for weeks. If you see a model is too slow then either a) decrease the data size, or b) just cut it short. Check also the `n_jobs` argument, in particular for the `cross_validate` function. Whatever you ended up doing, report what did you do, and why. This is life: many promising methods may be just too slow on the data.

The problem set is mostly about using the existing libraries, I just ask you to implement one algorithm, forward selection, yourself. We also follow the three-fold training-validation-testing protocol here.

Please submit a) your code (notebooks or whatever you are using) *and* b) the results in a final output form (html or pdf).

You are welcome to answer some of the questions on paper but please include the result as an image in your final file. Note that notebooks are just markdown documents (besides the code), and hence it's easy to include images.

Also, please stay focused with your solutions. Do not include dozens of pages of computer output (no one is willing to look at that many numbers). Focus on few relevant outputs only.

Working together is fun and useful but you have to submit your own work. Discussing the solutions and problems with your classmates is all right but do not copy-paste their solution! Please list all your collaborators below:

1.
2. ...

Rotten Tomatoes

Our first task is to load, clean and explore the Rotten Tomatoes movie reviews data. Please familiarize yourself a little bit with the webpage. Briefly, approved critics can write movie reviews, and evaluate the movie as "fresh" or "rotten". The webpage normally shows a short "quote" from each critic, and whether it was evaluated as fresh or rotten. You will work on these quotes below.

The central variables in *rotten-tomatoes.csv* are the following:

fresh evaluation: 'fresh' or 'rotten'

quote short version of the review

There are more variables like critic's name, date, and the link to IMDB.

1 Load data, and separate test data

Load data and split it into working and testing chunks. But before you begin: ensure you can save a dataframe in a format you can load back in afterwards. `pd.to_csv` is a good bet, but it has a lot of options which may screw up the way you read data. Ensure you can store data in a way you can read it back in correctly, including that missings remain missings.

1. create a tiny toy data frame that includes some numbers, strings, and missings. Save it and ensure you can reload it in the correct form.

Now you are good to go:

2. load the data (available on canvas: files/data/rotten-tomatoes.csv). **DO NOT LOOK AT IT**!
3. split the dataset into working-testing parts (80/20 or so). Note that *sklearn*'s `train_test_split` can easily handle dataframes. Just for your confirmation, ensure that the size of the working and testing data look reasonable.
4. now save the test data and *delete it from memory* (`del` statement is your friend).

2 Explore, clean and transform the data

Now when the test data is put aside, we can breath normally again and take a closer look how does the work data look like.

1. Take a look at a few lines of data (you may use `pd.sample` for this).
2. print out all variable names.
3. create a summary table (maybe more like a bullet list) where you print out the most important summary statistics for the most interesting variables. The most interesting facts you should present include: a) number of missings for *fresh* and *quote*; b) all different values for fresh/rotten evaluations; c) counts or percentages of these values; d) number of zero-length or only whitespace *quote*-s; e) minimum-maximum-average length of quotes (either in words, or in characters). (Can you do this as an one-liner?); f) how many reviews are in data multiple times. Feel free to add more figures you consider relevant.
4. Now when you have an overview what you have in data, clean it by removing all the inconsistencies the table reveals. We have to ensure that the central variables, *quote* and *fresh*, are not missing, and *quote* is not an empty string (or just contain spaces and such).

   I strongly recommend to do it as a standalone function because at the end you have to perform exactly the same cleaning operations with your test data too.
5. As a final step of data exploration, create a "naive" baseline model. As the baseline model I mean a model that does not look for any feature and predicts either *fresh* or *rotten*, whichever is more likely, for every case. Find accuracy for this model. This is our benchmark accuracy.

   (you don't have to program anything here, just compute the accuracy)

The data comes in textual form but our methods can only handle numbers. Hence we have to convert it into something that the models can digest. We go for the *Bag of Words* (BoW) representation. BOW represents each quote by the presence of words and ignores their order. A simple way to do the conversion is through `CountVectorizer` function in *sklearn.feature_ extraction.text* along the following lines:

```
vec = CountVectorizer(min_df, stop_words='english', binary=True)
X = vec.fit_transform(x)
```

A few comments may be needed here: `min_df` is the minimal number of times the word must be present in order to be taken into account. This is the prime way to scale down the data size when the full dataset becomes too slow. E.g. You may use values 1 for Naive Bayes and 20 for the logistic regression. `binary` means that `CountVectorizer` only records the binary presence, not the actual counts of words. Finally, `fit_transform` creates a sparse matrix, it is an excellent way to improve speed and memory consumption for all the methods where it works. You can force it to a dense matrix with `toarray()` attribute if needed.

And here is a small example:

```
import numpy as np
from sklearn.feature_extraction.text import CountVectorizer
a = ["Well done is better than well said",
     "God gave us the gift of life; it is up to us to give ourselves the gift of living well"]
                              # by Franklin, Voltaire
vec = CountVectorizer(binary=True)
x = vec.fit_transform(a)
# results in sparse matrix that is faster in general
## convert to a dense matrix and print the BoW-s:
print(x.toarray())
## see the words in the bag in the correct order:

## [[1 1 0 0 0 0 1 0 0 0 0 0 1 1 0 0 0 0 1]
##  [0 0 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 1 1]]

print(vec.get_feature_names())

## ['better', 'done', 'gave', 'gift', 'give', 'god', 'is', 'it', 'life', 'living', 'of', 'ourselves
```

6. convert data into BoW-s.

7. ensure you understand how the data is now stored. What is the shape of X? What are rows? What are columns? How can you find which word corresponds to the column 30 (Hint: check out `get_feature_names()` attribute).

3 Linear Models

But enough of preparation. Now let's do some work. In all the following model classes below you proceed broadly as follows: 1. loop over hyperparameters you are evaluating; 2. estimate the model involving hyperparameters; 3. cross-validate the model using prediction accuracy; 4. find the best model in this class and report it's accuracy.

3.1 Linear Regression

We start with OLS. As this includes no hyperparameters, no loops here. However, stock OLS does predict a number, not a class. If you coded fresh/rotten as 1/0, you can take the class as $\mathbb{1}(\hat{y} > 0.5)$.

1. estimate a linear regression model that includes all features in your data X.

2. implement a function that computes prediction accuracy based on this linear regression model. Implement it yourself, and implement it in a way that you can use it for cross-validation below.

3. cross-validate your model. Use *sklearn*'s `cross_validate` and supply your own accuracy function for scoring. Report the score.

3.2 Logistic regression

Well, linear regression is not really made for categorization kind of tasks. Logistic is. But is it any better?

1. Estimate a logistic regression model that includes all the features.

   Note: the stock logistic regression (`sklearn.linear_model.LogisticRegression`) by default uses regularization. Switch it off for now by setting `C = np.inf`.

2. cross-validate the model, and report the accuracy.

3.3 Forward selection

This data includes up to 20k features, so evaluating all possible models is way out of what we can do in this universe. Let's implement (i.e. don't use stock libraries) the forward selection procedure for Logistic Regression instead. Use 5-fold CV again to evaluate your models.
Note: even forward selection may be far too slow.

1. consult James et al. (2015, section 6.1), in particular page 207.

2. Create a series of 1-feature logistic regression models and pick the best one by CV. As before, use accuracy as the evaluation criterion.

   Note: always include the constant.

3. Pick the feature with the highest accuracy. This is your 1-feature model.

4. Repeat the procedure with more features until all features are included, or until your accuracy score shows a persistend downward trend (i.e. you are consistently overfitting).

5. How many features are in the best model? What is it's accuracy?

3.4 Ridge and Lasso

Next, let's move to regularization. In class we talked about two regularization methods: ridge ($\mathfrak{l}_2$-penalty) and lasso ($\mathfrak{l}_1$-penalty). We presented these in a linear regression context, but they work pretty much the same way for the logistic too (instead of *RSS*, we add penalty to log-likelihood). The task here is to find the best logistic regression model in terms of penalty ($\mathfrak{l}_1$ or $\mathfrak{l}_2$), and the penalty parameter λ.

1. consult James et al. (2015, section 6.2).

2. (re)read the documentation for `LogisticRegression`. In particular, understand how to enter the penalty parameter, and the penalty type.

3. write a loop over different penalty types and parameter values.

4. inside the loop, estimate the corresponding logistic regression, and find it's accuracy by CV.

5. report the best hyperparameter values and the related accuracy.

   Present your findings on a figure.

4 Naive Bayes

Now we are done with linear models. These are great models and will serve as a benchmark for the following ones. Unfortunately they are not designed for such a data. But Naive Bayes is made for exactly such cases. Let's figure it out.

Here we test two hyperparameters: `min_df` and `alpha`. The former belongs to `CountVectorizer` and tells what must the minimum number of times the word is present in the data be, see above. `alpha` is the NB smoother, the pseudo-count what we assume we have seen every single word even before we look at data.

1. read the documentation for `sklearn.naive_bayes.MultinomialNB`.
2. write a loop over different `df_min` and `alpha`.
3. inside the loop, estimate the corresponding NB model and CV it's accuracy.
4. report the best hyperparameter values and the related accuracy.

   Present your findings on a figure.

5 Trees and forests

Our final class of models are trees and related methods. We squeeze through decision trees, random forests, and boosting in a similar fashion as above.

5.1 Decision Trees

First, let's start with plain trees. Decision trees (`DecisionTreeClassifier`) has a large number of parameters to play with, I recommend to choose `min_impurity_decrease` (this is about the same as entropy gain at split) and `max_depth` (how deep nested trees we build). But you can choose something else if you wish.

1. Consult James James et al. (2015, ch 8.1)
2. Proceed as above: loop over the hyperparameters, cross-validate your accuracy, plot your results, and present the best model.

5.2 Random forests

Is forest better than the best tree? Find it out! `RandomForestClassifier` has a number of hyperparameters, you may consider, for instance, *n_ estimators*, *max_ depth* and *min_ impurity_ decrease*.

1. Consult James James et al. (2015, ch 8.2)
2. Do as before: loop over the hyperparameters, cross-validate your accuracy, plot your results, and present the best model.

5.3 AdaBoost

The final tree-related method is AdaBoost. It is an another ensemble method were a number of small trees are aggregated into a large forest. You can choose *n_ estimators* or *learning_ rate*, for instance, to optimize over.

1. Consult James James et al. (2015, ch 8.2.3)
2. . . . and repeat what you did earlier: CV for the best hyperparameters, plot and report.

6 Final things

Now you have walked through a whole load of methods, some of which did plain suck, and some of which worked pretty well.

1. Reflect what did you do and what did you find. Which methods were good? Which ones were bad? Can you explain why?

And the final-final thing: model performance on testing data.

1. Pick your favorite method. It is normally the one that gave you the best CV score, but in case that method caused other troubles (too slow, computer crashed, ...) you may opt for something else.
2. Fit the model using the cross-validated optimal hyperparameters using your complete work data (both training and validation). This is your best and final model.
3. Load your testing data. Clean it using exactly the same procedure (you made a function for this, right?) and transform it into BOW-s.

   Note: above I suggested using `vectorizer.fit_transform(quote)` function to create the BOW. Here I recommend to use `vectorizer.transform(quote)`. This is because we don't want to change the vocabulary (that's what the `fit`-part does), only to transform it into the BOW.
4. ...and *now* predict on testing data, and compute accuracy. This is your final performance measure!

   Congrats! You are done! And I *mean you are done*. Even if the final measure sucks, **don't go back to tinker with the models!** That's it. Submit, and get a life ☺

References

James, G., Witten, D., Hastie, T., Tibshirani, R., 2015. An Introduction to Statistical Learning with Applications in R. Springer.