classification

August 6, 2022

1 Classification

Let us add functions which would assist us in inputing the data to a model

First of all, let us remove all columns with more than 80% nulls

```
[ ]: nulls = X.isnull().sum().to_frame().reset_index().rename(columns={'index':␣
      ↪'column', 0:'nulls'})
     nulls['pc'] = nulls['nulls'] / len(X) * 100
     columns_to_drop = nulls[nulls['pc'] >= 80]['column'].to_list()
```

So we have 25.4K records in our training set;

0.05 of them to find the best way to fill missing data

0.3 for feature engineering, type of model, and 0.35 model tuning

```
[ ]: columns_to_drop = columns_to_drop + ['category', 'category_enc']
     X = clean_text_values(dataset_w_nutrients.drop(columns_to_drop, axis=1)).
      ↪set_index('idx')
     y = dataset_w_nutrients[['idx', 'category_enc']].
      ↪set_index('idx')['category_enc']
     X_train, X_test, y_train, y_test = model_selection.train_test_split(X, y,␣
      ↪test_size=0.2, random_state=1337, stratify=y)

     # 0.05 for missing data imputation
     X_rest, X_rest2, y_rest, y_rest2 = model_selection.train_test_split(X_train,␣
      ↪y_train, test_size=0.1, random_state=1337, stratify=y_train)
     X_mdi, X_coc, y_mdi, y_coc = model_selection.train_test_split(X_rest2, y_rest2,␣
      ↪test_size=0.5, random_state=1337, stratify=y_rest2)

     # 0.3 for feature engineering, resampling, and 0.35 model tuning
     X_rest, X_fe, y_rest, y_fe = model_selection.train_test_split(X_rest, y_rest,␣
      ↪test_size=1/3, random_state=1337, stratify=y_rest)
     X_rs, X_mt, y_rs, y_mt = model_selection.train_test_split(X_rest, y_rest,␣
      ↪test_size=0.5, random_state=1337, stratify=y_rest)
     X_mt, y_mt = pd.concat([X_mt, X_coc], axis=0), pd.concat([y_mt, y_coc], axis=0)
```

Initial feature engineering

First, let's clean the text columns.

We saw in the EDA that there are some sentences in the ingredients column, as well as numbers.

Let us look for those sentences again:

Let us extract a list of most common ingredients from each category to look for fishy strings and get some ideas for features:

```
[ ]: X_fe_with_cat = pd.concat([X_fe.reset_index(), pd.Series(le.
      ↪inverse_transform(y_fe))], axis=1).rename(columns={0: 'category'})
```

```
[ ]: X_fe_with_cat, top_15_words = get_ngrams_top_k(X_fe_with_cat, 'ingredients', 1,␣
      ↪15, True)
     # for cat, top in top_15_words.items():
     #     px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
      ↪words'.format(cat), height=400, width=500).show()
```

Let's do the same for 5-grams to identify common sentences:

```
[ ]: X_fe_with_cat, top_15_5grams = get_ngrams_top_k(X_fe_with_cat, 'ingredients',␣
      ↪5, 15, True)
     # for cat, top in top_15_5grams.items():
     #     px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
      ↪5-grams'.format(cat), height=400, width=500).show()
```

We can see that we have some bad words like 'or', 'and' etc (also numbers). Let's fix that:

```
[ ]: irrelevant_strings = ['and', 'contains', 'one', 'or', 'more', 'less', 'than',␣
      ↪'of', 'a', 'the', 'following'] + [i for i in range(0, 1000)]
     for word in irrelevant_strings:
         X_fe['ingredients'] = X_fe['ingredients'].str.replace(" {} ".format(word),␣
      ↪" ")
         X_fe_with_cat['ingredients'] = X_fe_with_cat['ingredients'].str.replace("␣
      ↪{} ".format(word), " ")
```

Let's see the new top words and 5-grams:

```
[ ]: X_fe_with_cat, top_15_words = get_ngrams_top_k(X_fe_with_cat, 'ingredients', 1,␣
      ↪15, True)
     # for cat, top in top_15_words.items():
     #     px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
      ↪words'.format(cat), height=400, width=500).show()
```

```
[ ]: X_fe_with_cat, top_15_5grams = get_ngrams_top_k(X_fe_with_cat, 'ingredients',␣
      ↪5, 15, True)
     # for cat, top in top_15_5grams.items():
     #     px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
      ↪5-grams'.format(cat), height=400, width=500).show()
```

Numbers and sentences eliminated!

Let’s move on to create features from the ingredients:

```
[ ]: ingredients_strings = np.unique(np.concatenate([top_15_words[cat]['ngram'].
     ↪values for cat in top_15_words.keys()], axis=0))
```

Let’s look for common words in the description

```
[ ]: X_fe_with_cat, top_15_desc_words = get_ngrams_top_k(X_fe_with_cat,␣
     ↪'description', 1, 15)
     # for cat, top in top_15_desc_words.items():
     #     px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
     ↪words'.format(cat), height=400, width=500).show()
```

This time we’ll compile the strings manually because we don’t need all of them:

```
[ ]: description_strings = np.array([
         'roasted', 'mix', 'almonds', 'corn', 'nuts', 'chocolate', 'trail',␣
     ↪'cashews', 'salt', 'covered', 'dark', 'milk', 'caramel', 'bar',
         'truffles', 'candy', 'fruit', 'sour', 'gumm', 'jell', 'candies', 'cherry',␣
     ↪'chewy', 'chips', 'potato', 'tortilla', 'kettle', 'pretzel',
         'cookie', 'chip', 'sugar', 'sandwich', 'butter', 'oatmeal', 'vanilla',␣
     ↪'fudge', 'cake', 'pie', 'donut', 'cupcakes', 'cheesecake', 'bakery',
         'brownie'
     ])
```

Let’s do the same for the brands:

```
[ ]: X_fe_with_cat, top_15_brand_words = get_ngrams_top_k(X_fe_with_cat, 'brand', 1,␣
     ↪15)
     # for cat, top in top_15_brand_words.items():
     #     px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
     ↪words'.format(cat), height=400, width=500).show()
```

```
[ ]: brand_strings = np.array([
         'snack', 'chocola', 'candy', 'candies', 'chips', 'potato', 'bake',␣
     ↪'baking', 'biscuit', 'cookie', 'jelly', 'nuts'
     ])
```

And do the same for the household_serving_fulltext:

```
[ ]: X_fe_with_cat, top_15_household_serving_fulltext_words =␣
     ↪get_ngrams_top_k(X_fe_with_cat, 'household_serving_fulltext', 1, 15)
     # for cat, top in top_15_household_serving_fulltext_words.items():
     #     px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
     ↪words'.format(cat), height=400, width=500).show()
```

```
[ ]: household_serving_fulltext_strings = np.array([
         'cup', 'onz', 'tbsp', 'pieces', 'about', 'package', 'grm', 'bag', 'bar',␣
       ↪'piece', 'squares', 'pcs', 'pouch', 'chips', 'pretzels', 'cookie',
         'wafers', 'crackers','slice', 'pie', 'cake', 'cupcake', 'donut'
     ])
```

Let us do the same for bi-grams:

```
[ ]: X_fe_with_cat, top_7_ingredients_bigrams = get_ngrams_top_k(X_fe_with_cat,␣
       ↪'ingredients', 2, 7, True)
     for cat, top in top_7_ingredients_bigrams.items():
         #px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
       ↪words'.format(cat), height=400, width=500).show()
         ingredients_strings = np.concatenate([ingredients_strings, top['ngram'].
       ↪values], axis=0)
```

```
[ ]: X_fe_with_cat, top_7_description_bigrams = get_ngrams_top_k(X_fe_with_cat,␣
       ↪'description', 2, 7, True)
     for cat, top in top_7_description_bigrams.items():
         #px.histogram(top, orientation='h', y='ngram', x='count', title='{} top␣
       ↪words'.format(cat), height=400, width=500).show()
         description_strings = np.concatenate([description_strings, top['ngram'].
       ↪values], axis=0)
```

Let us create a proper dictionary for convinience later:

```
[ ]: feature_strings_dict = {
         'ingredients': np.unique(ingredients_strings),
         'description': np.unique(description_strings),
         'brand': np.unique(brand_strings),
         'household_serving_fulltext': np.unique(household_serving_fulltext_strings)
     }
```

We also created a function to fix some brand values (which were manually picked from the X_fe dataset).

After the initial feature engineering, let us choose how we will impute our values

We'll create a pipeline which first imputes the missing values with various methods, and then cross validates the results of a vanilla XGB classifier. The methods used are comprised of simple imputations including the mean, median values or filling all missing values with 0, and more complex imputations using linear and trees regressions with various parameters to impute the missing values. Using balanced accuracy score to account for class imbalance, as we have yet to address it.

```
[ ]: X_mdi_fe = fe(X_mdi, feature_strings_dict)
```

```
[ ]: mdi_pipeline_steps = {'imputer':['simple_imp', 'iterative_imp'],
                           'classifier':['xgbc']}
```

```
# fill parameters to be searched in this dict
mdi_all_param_grids = {'xgbc':{'object':XGBClassifier(random_state=1337),
                          'use_label_encoder': [False],
                          'objective': ['multi:softmax'],
                          'eval_metric': ['mlogloss']
                        , 'tree_method': ['gpu_hist']
                         },
                   'simple_imp':{
                        'object': SimpleImputer(),
                        'strategy':['mean', 'median', 'constant']
                        },
                   'iterative_imp':{
                        'object':IterativeImputer(random_state=1337),
                        'estimator':[linear_model.
 ↪ElasticNet(random_state=1337)],
                        'max_iter': [200],
                        'skip_complete': [True],
                        'initial_strategy': ['mean', 'median', 'constant'],
                        'imputation_order': ['ascending', 'descending',␣
 ↪'random']
                        }
                  }

mdi_param_grids_list = make_param_grids(mdi_pipeline_steps, mdi_all_param_grids)
mdi_pipe = Pipeline(steps=[('imputer', SimpleImputer()), ('classifier',␣
 ↪XGBClassifier())])
mdi_grid = model_selection.GridSearchCV(mdi_pipe, param_grid =␣
 ↪mdi_param_grids_list, scoring='balanced_accuracy', cv=10, verbose=15)
mdi_grid.fit(X_mdi_fe, y_mdi)
```

```
[ ]: mdi_grid.best_params_
```

```
[ ]: pd.DataFrame(mdi_grid.cv_results_).sort_values('rank_test_score').
      ↪head(5)[['params', 'mean_test_score']].values
```

Result is:

IterativeImputer(estimator=ElasticNet(random_state=1337), imputation_order='descending', initial_strategy='median', max_iter=200, random_state=1337, skip_complete=True)

On to resampling.

We won't test only downsampling because we want to get a balanced dataset, and the minority classes have too few samples.

We'll test several strategies - Random upsampling, random upsampling with downsampling, SMOTE upsampling, SMOTE upsampling with downsampling, KMeans SMOTE oversampling, KMeans SMOTE oversampling with downsampling.

For downsampling we will test random downsampling, and no downsampling.

Each strategy which involves downsampling will test 10 values - 0.95, 0.9, 0.85, 0.8, 0.75, 0.7, 0.65, 0.6, 0.55, 0.5 of the largest class

The random upsampling strategies will also test shrinkage values - 0, 1, 2, 3

The SMOTE strategies will test different numbers of k_neighbors - 5, 10, 15

In total we get 77 strategies to choose from.

In this case we will also use a vanilla XGBoost to compare out results (this time using accuracy instead of balanced accuracy)

[ ]:
```
chosen_imputer = IterativeImputer(estimator=linear_model.
 ↪ElasticNet(random_state=1337),
                           imputation_order='descending', initial_strategy='mean',
                           max_iter=200, random_state=1337, skip_complete=True)
chosen_imputer.fit(X_mdi_fe)
```

[ ]:
```
IterativeImputer(estimator=ElasticNet(random_state=1337),
                 imputation_order='descending', max_iter=200, random_state=1337,
                 skip_complete=True)
```

[ ]:
```
X_rs_fe = fe(X_rs, feature_strings_dict)
X_rs_imputed = chosen_imputer.transform(X_rs_fe)
```

[ ]:
```
# Had to give up lambdas to recognize functions after CV :(

# # Creates lambas to feed into the strategies argument of the downsamplers␣
 ↪(because of the CV the numbers are not constant)
# def get_strategy_lambda(i):
#     return lambda y: {j: round(max(np.bincount(y)) * i * 0.1) for j in␣
 ↪range(0,6)}
# strategy_lambdas = [get_strategy_lambda(i) for i in range(6, 10)]
strategy_funcs = []
def_strategy_function_exec_code = """def strategy_{}(y): return {{j:␣
 ↪round(max(np.bincount(y)) * {} * 0.05) for j in range(0,6)}}"""
for i in range(10,20):
  exec(def_strategy_function_exec_code.format(round(i * 5), i))
  exec("""strategy_funcs.append(strategy_{})""".format(i * 5))


# Had to use these step names becasue imblearn uses a custom pipe constructor.
# found this out after I finished these dicts and almost cried in fear of a␣
 ↪much more complicated CV
rs_pipeline_steps = {
     'randomoversampler': ['random_us', 'smote_us'],
     'randomundersampler': ['no_ds', 'random_ds'],
     'xgbclassifier': ['xgbc']
}

rs_all_param_grids = {'xgbc': {'object':XGBClassifier(random_state=1337),
                          'use_label_encoder': [False],
                          'objective': ['multi:softmax'],
                          'eval_metric': ['mlogloss']
                        , 'tree_method': ['gpu_hist']
                         },
                   'random_us': {
                        'object': over_sampling.
 ↪RandomOverSampler(random_state=1337),
                        'shrinkage': [0, 1, 2, 3]
                        },
                   'smote_us': {
                        'object': over_sampling.SMOTE(random_state=1337),
                        'k_neighbors': [5, 10, 15]
                        },
                   'no_ds': {
                        'object': None
                        },
                   'random_ds': {
                        'object': under_sampling.
 ↪RandomUnderSampler(random_state=1337),
                        'sampling_strategy': strategy_funcs
                        }
                  }

rs_param_grids_list = make_param_grids(rs_pipeline_steps, rs_all_param_grids)
# rs_pipe = Pipeline(steps=[('up_sample', None), ('down_sample', None),␣
 ↪('classifier', None)])
rs_pipe = pipeline.make_pipeline(over_sampling.RandomOverSampler(),␣
 ↪under_sampling.RandomUnderSampler(), XGBClassifier()) # imblearn pipe
rs_grid = model_selection.GridSearchCV(rs_pipe, param_grid =␣
 ↪rs_param_grids_list, scoring='accuracy', cv=10, verbose=15)
rs_grid.fit(X_rs_imputed, y_rs)
```

Results are: SMOTE(k_neighbors=10, random_state=1337)

RandomUnderSampler(random_state=1337, sampling_strategy=<function strategy_75 at 0x7fec5bb6def0>)

So, SMOTE with k_neighbors=10 and RandomUnderSampler with a downsampling to 75% of the largest class.

[ ]:
```
pd.options.mode.chained_assignment = None

brands_for_column_values = X_fe[['brand']] # Take brands from FE portion of␣
 ↪dataset
```

```
brands_column_values = get_column_values(brands_for_column_values)
brands = X_fe[['brand']]
ok_brands = brands_column_values['brand'][brands_column_values['brand']['pc']␣
↪>= 0.1]['value'].values
brands_mask = ~brands['brand'].isin(ok_brands)
brands.loc[brands_mask, 'brand'] = 'other'
brands_ohe, _ = get_oh_encoder(brands)
brands_ohe.fit(brands)
```

```
[ ]: OneHotEncoder(categories=[['7eleven inc', 'abimar foods inc', 'ahold usa inc',
                                'aldibenner company', 'american halal company inc',
                                'american importing co inc', 'archer farms',
                                'back to nature foods company llc',
                                'bee international inc', 'bergin nut company inc',
                                'best choice', 'better made snack foods inc',
                                'big y foods inc', 'bimbo bakeries usa inc',
                                'bjs wholesale club corporate brands',
                                'blue diamond growers', 'brads raw chips',
                                'brookshire grocery company',
                                'california flavored nuts', 'candyrific llc',
                                'cape cod potato chips inc', 'charms company',
                                'cibo vita inc', 'creative natural products inc',
                                'creative snacks co llc',
                                'csm bakery products na inc', 'cvs pharmacy inc',
                                'dawn food products inc', 'delhaize america inc',
                                'demets candy company', …]],
                  handle_unknown='ignore', sparse=False)
```

Let us remove highly correlated features

Now let us write a function which ties the dataset together, before moving to high correlated columns removal

```
[ ]: def get_numerical_df(df, feature_strings_dict, brands_ohe):
       df = fix_brands(df)
       brands = df[['brand']]
       brands_oh = brands_ohe.transform(brands)
       df_fe = fe(df, feature_strings_dict)
       numerical_df = np.concatenate([brands_oh, df_fe], axis=1)
       return numerical_df
```

```
[ ]: # Refit chosen imputer to the new columns
     X_mdi_numerical = get_numerical_df(X_mdi, feature_strings_dict, brands_ohe)
     chosen_imputer = IterativeImputer(estimator=linear_model.
      ↪ElasticNet(random_state=1337),
                                   imputation_order='descending', initial_strategy='mean',
                                   max_iter=200, random_state=1337, skip_complete=True)
     chosen_imputer.fit(X_mdi_numerical)
```

```
[ ]: IterativeImputer(estimator=ElasticNet(random_state=1337),
                      imputation_order='descending', max_iter=200, random_state=1337,
                      skip_complete=True)
```

```
[ ]: X_fe_numerical = get_numerical_df(X_fe, feature_strings_dict, brands_ohe)
     X_fe_numerical_imputed = chosen_imputer.transform(X_fe_numerical)
```

Let's use CV to find best value to consider a high correlation:

```
[ ]: def strategy_75(y):
       return {j: round(max(np.bincount(y)) * 15 * 0.05) for j in range(0,6)}

     chosen_os = over_sampling.SMOTE(k_neighbors=10, random_state=1337)
     chosen_us = under_sampling.RandomUnderSampler(random_state=1337,␣
      ↪sampling_strategy=strategy_75)
     vanilla_xgbc = XGBClassifier(eval_metric='mlogloss', objective='multi:softmax',
                                  random_state=1337, tree_method='gpu_hist',
                                  use_label_encoder=False)
```

```
[ ]: corr_pipe = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
     cv_scores = {}
     for corr in [0.6, 0.7, 0.8, 0.9]:
       X_fe_numerical_df = pd.DataFrame(X_fe_numerical_imputed)
       X_fe_corr = X_fe_numerical_df.corr()
       upper_tri = X_fe_corr.where(np.triu(np.ones(X_fe_corr.shape), k=1).
      ↪astype(bool))
       corr_columns_to_drop = [column for column in upper_tri.columns if␣
      ↪any(upper_tri[column] > corr)]
       X_fe_numerical_corr = X_fe_numerical_df.drop(corr_columns_to_drop, axis=1).
      ↪values
       cv_scores[corr] = model_selection.cross_val_score(corr_pipe,␣
      ↪X_fe_numerical_corr, y_fe, cv=10,
                                                         verbose=15, n_jobs=-1).
      ↪mean()
     max_score = max(cv_scores, key=cv_scores.get)
     max_score
```

So we will remove all columns with correlation above the chosen threshold.

```
[ ]: X_fe_numerical_df = pd.DataFrame(X_fe_numerical_imputed)
     X_fe_corr = X_fe_numerical_df.corr()
     upper_tri = X_fe_corr.where(np.triu(np.ones(X_fe_corr.shape), k=1).astype(bool))
     corr_columns_to_drop = [column for column in upper_tri.columns if␣
      ↪any(upper_tri[column] > max_score)]
```

```
[ ]: def get_final_matrix(X, feature_strings_dict, brands_ohe, corr_columns_to_drop,␣
      ↪chosen_imputer):
```

```
    numerical_X = get_numerical_df(X, feature_strings_dict, brands_ohe)
    X_imputed_df = pd.DataFrame(chosen_imputer.transform(numerical_X))
    final_matrix = X_imputed_df.drop(corr_columns_to_drop, axis=1).values
    return final_matrix
```

[ ]:
```
X_mt_numerical = get_final_matrix(X_mt, feature_strings_dict, brands_ohe,␣
 ↪corr_columns_to_drop, chosen_imputer)
```

Now let us tune our XGB Classifier one step at a time, starting with the number of trees for the default learning rate.

[ ]:
```
mt_pipeline_steps_ne = {
     'xgbclassifier': ['xgbc']
}

mt_all_param_grids_ne = {'xgbc': {'object':XGBClassifier(random_state=1337),
                          'n_estimators': [100 + i * 10 for i in range(0, 41)],
                          'use_label_encoder': [False],
                          'objective': ['multi:softmax'],
                          'eval_metric': ['mlogloss']
                        , 'tree_method': ['gpu_hist']
                         }
                  }

mt_param_grids_list_ne = make_param_grids(mt_pipeline_steps_ne,␣
 ↪mt_all_param_grids_ne)
mt_pipe_ne = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
mt_grid_ne = model_selection.GridSearchCV(mt_pipe_ne, param_grid =␣
 ↪mt_param_grids_list_ne, scoring='accuracy', cv=10, verbose=15)
mt_grid_ne.fit(X_mt_numerical, y_mt)
```

[ ]:
```
mt_grid_ne.best_params_['xgbclassifier__n_estimators']
```

Now let's tune other parameters:

[ ]:
```
mt_pipeline_steps_mdcw = {
     'xgbclassifier': ['xgbc']
}

mt_all_param_grids_mdcw = {'xgbc': {'object':XGBClassifier(random_state=1337),
                          'n_estimators': [mt_grid_ne.
 ↪best_params_['xgbclassifier__n_estimators']],
                          'max_depth': [i for i in range(3, 13)],
                          'min_child_weight': [i for i in range(1, 7)],
                          'use_label_encoder': [False],
                          'objective': ['multi:softmax'],
                           'eval_metric': ['mlogloss']
                         , 'tree_method': ['gpu_hist']
                          }
                  }

mt_param_grids_list_mdcw = make_param_grids(mt_pipeline_steps_mdcw,␣
 ↪mt_all_param_grids_mdcw)
mt_pipe_mdcw = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
mt_grid_mdcw = model_selection.GridSearchCV(mt_pipe_mdcw, param_grid =␣
 ↪mt_param_grids_list_mdcw, scoring='accuracy', cv=10, verbose=15)
mt_grid_mdcw.fit(X_mt_numerical, y_mt)
```

```
[ ]: mt_grid_mdcw.best_params_
```

```
[ ]: print(mt_grid_mdcw.best_params_['xgbclassifier__max_depth'])
     print(mt_grid_mdcw.best_params_['xgbclassifier__min_child_weight'])
```

```
[ ]: mt_pipeline_steps_gam = {
          'xgbclassifier': ['xgbc']
     }

     mt_all_param_grids_gam = {'xgbc': {'object':XGBClassifier(random_state=1337),
                                'n_estimators': [mt_grid_ne.
      ↪best_params_['xgbclassifier__n_estimators']],
                                'max_depth': [mt_grid_mdcw.
      ↪best_params_['xgbclassifier__max_depth']],
                                'min_child_weight': [mt_grid_mdcw.
      ↪best_params_['xgbclassifier__min_child_weight']],
                                'gamma': [i / 10.0 for i in range(0, 5)],
                                'use_label_encoder': [False],
                                'objective': ['multi:softmax'],
                                'eval_metric': ['mlogloss']
                              , 'tree_method': ['gpu_hist']
                               }
                       }

     mt_param_grids_list_gam = make_param_grids(mt_pipeline_steps_gam,␣
      ↪mt_all_param_grids_gam)
     mt_pipe_gam = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
     mt_grid_gam = model_selection.GridSearchCV(mt_pipe_gam, param_grid =␣
      ↪mt_param_grids_list_gam, scoring='accuracy', cv=10, verbose=15)
     mt_grid_gam.fit(X_mt_numerical, y_mt)
```

```
[ ]: mt_grid_gam.best_params_['xgbclassifier__gamma']
```

On to check subsample and colsample_bytree.

We will first check with 0.1 intervals, and then 0.05 intervals around the chosen values.

```
[ ]: mt_pipeline_steps_sscs = {
         'xgbclassifier': ['xgbc']
     }

     mt_all_param_grids_sscs = {'xgbc': {'object':XGBClassifier(random_state=1337),
                                 'n_estimators': [mt_grid_ne.
       ↪best_params_['xgbclassifier__n_estimators']],
                                 'max_depth': [mt_grid_mdcw.
       ↪best_params_['xgbclassifier__max_depth']],
                                 'min_child_weight': [mt_grid_mdcw.
       ↪best_params_['xgbclassifier__min_child_weight']],
                                 'gamma': [mt_grid_gam.
       ↪best_params_['xgbclassifier__gamma']],
                                 'subsample':[i/10.0 for i in range(6,11)],
                                 'colsample_bytree':[i/10.0 for i in range(6,11)],
                                 'use_label_encoder': [False],
                                 'objective': ['multi:softmax'],
                                 'eval_metric': ['mlogloss']
                               , 'tree_method': ['gpu_hist']
                                }
                       }

     mt_param_grids_list_sscs = make_param_grids(mt_pipeline_steps_sscs,␣
       ↪mt_all_param_grids_sscs)
     mt_pipe_sscs = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
     mt_grid_sscs = model_selection.GridSearchCV(mt_pipe_sscs, param_grid =␣
       ↪mt_param_grids_list_sscs, scoring='accuracy', cv=10, verbose=15)
     mt_grid_sscs.fit(X_mt_numerical, y_mt)
```

```
[ ]: ss = mt_grid_sscs.best_params_['xgbclassifier__subsample']
     csbt = mt_grid_sscs.best_params_['xgbclassifier__colsample_bytree']
     print(ss)
     print(csbt)
```

```
[ ]: ss = mt_grid_sscs.best_params_['xgbclassifier__subsample']
     csbt = mt_grid_sscs.best_params_['xgbclassifier__colsample_bytree']
     print(ss)
     print(csbt)
```

```
[ ]: mt_pipeline_steps_sscs2 = {
         'xgbclassifier': ['xgbc']
     }

     mt_all_param_grids_sscs2 = {'xgbc': {'object':XGBClassifier(random_state=1337),
                              'n_estimators': [mt_grid_ne.
  ↪best_params_['xgbclassifier__n_estimators']],
                              'max_depth': [mt_grid_mdcw.
  ↪best_params_['xgbclassifier__max_depth']],
                              'min_child_weight': [mt_grid_mdcw.
  ↪best_params_['xgbclassifier__min_child_weight']],
                              'subsample':[ss - 0.05, ss, ss + 0.05],
                              'colsample_bytree':[csbt - 0.05, csbt, csbt + 0.5],
                              'use_label_encoder': [False],
                              'objective': ['multi:softmax'],
                              'eval_metric': ['mlogloss']
                            , 'tree_method': ['gpu_hist']
                             }
                     }

mt_param_grids_list_sscs2 = make_param_grids(mt_pipeline_steps_sscs2,␣
  ↪mt_all_param_grids_sscs2)
mt_pipe_sscs2 = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
mt_grid_sscs2 = model_selection.GridSearchCV(mt_pipe_sscs2, param_grid =␣
  ↪mt_param_grids_list_sscs2, scoring='accuracy', cv=10, verbose=15)
mt_grid_sscs2.fit(X_mt_numerical, y_mt)
```

[ ]:
```
mt_grid_sscs2.best_params_['xgbclassifier__subsample']
```

Now for regularization parameters:

[ ]:
```
mt_pipeline_steps_reg = {
     'xgbclassifier': ['xgbc']
}

mt_all_param_grids_reg = {'xgbc': {'object':XGBClassifier(random_state=1337),
                              'n_estimators': [mt_grid_ne.
  ↪best_params_['xgbclassifier__n_estimators']],
                              'max_depth': [mt_grid_mdcw.
  ↪best_params_['xgbclassifier__max_depth']],
                              'min_child_weight': [mt_grid_mdcw.
  ↪best_params_['xgbclassifier__min_child_weight']],
                              'subsample':[mt_grid_sscs2.
  ↪best_params_['xgbclassifier__subsample']],
                              'colsample_bytree':[mt_grid_sscs2.
  ↪best_params_['xgbclassifier__colsample_bytree']],
                              'alpha': [0, 1e-5, 1e-2, 0.1, 1],
                              'lambda': [0, 1e-5, 1e-2, 0.1, 1],
                              'use_label_encoder': [False],
                              'objective': ['multi:softmax'],
                              'eval_metric': ['mlogloss']
```

```
                         , 'tree_method': ['gpu_hist']
                          }
                  }

mt_param_grids_list_reg = make_param_grids(mt_pipeline_steps_reg,␣
 ↪mt_all_param_grids_reg)
mt_pipe_reg = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_xgbc)
mt_grid_reg = model_selection.GridSearchCV(mt_pipe_reg, param_grid =␣
 ↪mt_param_grids_list_reg, scoring='accuracy', cv=10, verbose=15)
mt_grid_reg.fit(X_mt_numerical, y_mt)
```

[ ]:
```
mt_grid_reg.best_params_
```

For some reason tuning the learning rate parameter didn't work (got exact same results in CV).

So now we have out XGB classifier!

XGBClassifier(alpha=0, colsample_bytree=0.8, eval_metric='mlogloss', lambda=0, max_depth=12, min_child_weight=3, n_estimators=250, objective='multi:softmax', random_state=1337, subsample=1.0, tree_method='gpu_hist', use_label_encoder=False)

Let us move on to choosing a RF classifier using CV:

[ ]:
```
mt_pipeline_steps_rf = {
     'randomforestclassifier': ['rf']
}

mt_all_param_grids_rf = {'rf': {'object':ensemble.
 ↪RandomForestClassifier(random_state=1337),
                         'n_estimators': [100 + i * 50 for i in range(0, 9)],
                         'criterion': ['entropy'],
                         'max_features': ['sqrt'],
                         'min_samples_split': [2, 3, 4, 5]
                        }
                  }

mt_param_grids_list_rf = make_param_grids(mt_pipeline_steps_rf,␣
 ↪mt_all_param_grids_rf)
mt_pipe_rf = pipeline.make_pipeline(chosen_os, chosen_us, vanilla_rf)
mt_grid_rf = model_selection.GridSearchCV(mt_pipe_rf, param_grid =␣
 ↪mt_param_grids_list_rf, scoring='accuracy', cv=10, verbose=15)
mt_grid_rf.fit(X_mt_numerical, y_mt)
```

So, our chosen RF classifier is is:

RandomForestClassifier(criterion='entropy', n_estimators=200, random_state=1337)

Let su move on to a SVM classifier:

```
[ ]: mt_pipeline_steps_svm = {
         'svc': ['svc']
     }

     mt_all_param_grids_svm = {'svc': {'object': svm.SVC(),
                                       'C': [10000, 100000, 1000000],
                                       'kernel': ['rbf', 'poly']
                                      }
                              }

     mt_param_grids_list_svm = make_param_grids(mt_pipeline_steps_svm,␣
       ↪mt_all_param_grids_svm)
     mt_pipe_svm = pipeline.make_pipeline(chosen_os, chosen_us, svm.SVC())
     mt_grid_svm = model_selection.GridSearchCV(mt_pipe_svm, param_grid =␣
       ↪mt_param_grids_list_svm, scoring='accuracy', cv=5, verbose=15)
     mt_grid_svm.fit(X_mt_numerical, y_mt)
```

The result is:

SVC(C=100000)

So now we have our 2 models and we can finally get to trying them on the test set!

```
[ ]: # Prepare estimators:
     xgbc = XGBClassifier(alpha=0, colsample_bytree=0.8, eval_metric='mlogloss',␣
       ↪reg_lambda=0,
                   max_depth=12, min_child_weight=3, n_estimators=250,
                   objective='multi:softmax', random_state=1337, subsample=1.0,
                   tree_method='gpu_hist', use_label_encoder=False)
     rfc = ensemble.RandomForestClassifier(criterion='entropy', n_estimators=200,␣
       ↪random_state=1337)
     svmc = svm.SVC(C=100000)

     estimators_dict = {
         'xgbc': pipeline.make_pipeline(chosen_os, chosen_us, xgbc),
         'rfc': pipeline.make_pipeline(chosen_os, chosen_us, rfc),
         'svmc': pipeline.make_pipeline(chosen_os, chosen_us, svmc)
     }

     # Prepare test_data:
     X_train_numerical = get_final_matrix(X_train, feature_strings_dict, brands_ohe,␣
       ↪corr_columns_to_drop, chosen_imputer)

     # Fit
     for name, estimator in estimators_dict.items():
         print("Fitting {}".format(name))
         estimator.fit(X_train_numerical, y_train)
         print("Finished fitting {}".format(name))
```

```
Fitting xgbc
Finished fitting xgbc
Fitting rfc
Finished fitting rfc
Fitting svmc
Finished fitting svmc
```

Let us compute the score on the test set:

```
[ ]: accs = {}
     X_test_numerical = get_final_matrix(X_test, feature_strings_dict, brands_ohe,␣
      ↪corr_columns_to_drop, chosen_imputer)
     for name, estimator in estimators_dict.items():
       print("Predicting {}".format(name))
       y_pred = estimator.predict(X_test_numerical)
       accs[name] = metrics.accuracy_score(y_test, y_pred)
       print("Finished predicting {}".format(name))
     print(accs)
```

```
Predicting xgbc
Finished predicting xgbc
Predicting rfc
Finished predicting rfc
Predicting svmc
Finished predicting svmc
{'xgbc': 0.9310344827586207, 'rfc': 0.9266257282317745, 'svmc':
0.9044245000787278}
```

Not too bad. Now let us train our classifiers on all the data and make predictions for the test csv.

```
[ ]: X_numerical = get_final_matrix(X, feature_strings_dict, brands_ohe,␣
      ↪corr_columns_to_drop, chosen_imputer)
     final_estimators_dict = {
         'xgbc': pipeline.make_pipeline(chosen_os, chosen_us, xgbc),
         'rfc': pipeline.make_pipeline(chosen_os, chosen_us, rfc),
         'svmc': pipeline.make_pipeline(chosen_os, chosen_us, svmc)
     }
     for name, estimator in final_estimators_dict.items():
         print("Fitting {}".format(name))
         estimator.fit(X_numerical, y)
         print("Finished fitting {}".format(name))
```

```
Fitting xgbc
Finished fitting xgbc
Fitting rfc
Finished fitting rfc
Fitting svmc
Finished fitting svmc
```

```
[ ]: test = pd.read_csv('drive/MyDrive/Final_Project-orav94/data/food_test.csv')
     joined_test = pd.merge(test, nutrients, how='left', on='idx').
      ↪drop(['nutrient_id', 'unit_name'], axis=1)
     pivoted_test = pd.pivot_table(
         joined_test,
         values='amount',
         index='idx',
         columns='name')

     test_w_nutrients = pd.merge(test, pivoted_test, how='left', on='idx')
     test_ids = test_w_nutrients[['idx']]
     excl = ['alcohol_ethyl_g', 'energy_kj', 'folic_acid_ug', 'molybdenum_mo_ug',␣
      ↪'starch_g']
     test_col_drop = [e for e in columns_to_drop[:len(columns_to_drop)-2] if e not␣
      ↪in excl]
     test_final = clean_text_values(test_w_nutrients.drop(test_col_drop, axis=1)).
      ↪set_index('idx')
     test_numerical = get_final_matrix(test_final, feature_strings_dict, brands_ohe,␣
      ↪corr_columns_to_drop, chosen_imputer)

     preds = {}
     for name, estimator in final_estimators_dict.items():
         print("Predicting {}".format(name))
         preds[name] = estimator.predict(test_numerical)
         print("Finished predicting {}".format(name))
```

```
Predicting xgbc
Finished predicting xgbc
Predicting rfc
Finished predicting rfc
Predicting svmc
Finished predicting svmc
```

```
[ ]: def decode(n):
       decode_dict = {
           0: 'cakes_cupcakes_snack_cakes', 1: 'candy', 2: 'chocolate', 3:
      ↪'cookies_biscuits',
           4: 'popcorn_peanuts_seeds_related_snacks', 5:'chips_pretzels_snacks'
       }
       return decode_dict[n]

     preds_dfs = {name: pd.concat([test_ids, pd.DataFrame(pred)], axis=1) for name,␣
      ↪pred in preds.items()}
     for name in preds_dfs.keys():
       preds_dfs[name] = preds_dfs[name].rename(columns={0: 'category_enc'})
       preds_dfs[name]['pred'] = preds_dfs[name]['category_enc'].apply(decode)
       preds_dfs[name] = preds_dfs[name].drop(['category_enc'], axis=1)
```

```
 preds_dfs[name].to_csv(path_or_buf='data/pred_{}.csv'.format(name),␣
↪columns=['idx', 'pred'], index=False)
```