

STAT 441 Project Final Report

Maisha Thasin, Joseph Wang (Group 25, undergraduate)

Winter 2023

Summary

- Do this last

Introduction

- Talk about the issue (churn) and its applications
- Describe the dataset
- Introduce the research question
- EDA

Dataset

The Telco Customer Churn dataset, provided by IBM, contains information about a fictional telecommunications (telco) company which provided home phone and internet services to 7043 customers. The dataset provides demographic and business-related metrics for each customer, as well as identifying whether the customer switched providers (customer churn).

The dataset can be downloaded from the following link: https://accelerator.ca.analytics.ibm.com/bi/?perspective=authoring&pathRef=.public_folders%2FIBM%2BAccelerator%2BCatalog%2FContent%2FDAT00067

The dataset contains **33** variables for **7043** observations, but not all variables are fit to be predictive features. We removed columns relating to unique IDs, geographical information, and dashboarding aggregation, as well as “duplicate” columns (those that are identical to another column except for formatting), and columns related to the response (such as the churn reason and predicted lifetime value to the company).

We are then left with **19** features: Gender, Senior Citizen, Partner, Dependents, Tenure Months (integer/numeric), Phone Service, Multiple Lines, Internet Service, Online Security, Online Backup, Device Protection Plan, Tech Support, Streaming TV, Streaming Movies, Contract, Paperless Billing, Payment method, Monthly Charge (float/numeric), and Total Charges (float/numeric). Each feature is categorical except those specified otherwise. The descriptions of the features can be found at <https://community.ibm.com/community/user/businessanalytics/blogs/steven-macko/2019/07/11/telco-customer-churn-1113>.

Finally, the response variable is Churn Value, binary on whether the customer left the company this quarter.

Methods

Preprocessing

In our initial investigation, we identified which columns of the dataset we were going to use as features, as well as the label. We then determined the type of each feature, with all but three being categorical (including binary features), and the remainder being of unbounded numeric type. We then removed incomplete rows, leaving 7032 observations.

The preprocessing steps, including the train/test split, were done in a unified manner so that each model would be trained and tested on the exact same data, so that accurate comparisons can be made from the results. For use in a supervised classification task, we also introduced some additional preprocessing steps (not all of which are used by every model).

The first optional step is Synthetic Minority Over-sampling Technique (SMOTE) (Chawla et al. 2002). SMOTE is an over-sampling method intended to reduce the bias towards the majority class in classification models by artificially

creating new datapoints in the minority class. The authors note that “the combination of SMOTE and under-sampling also performs better... than the methods that could directly handle the skewed class distribution”. The minority class in our dataset represents approximately 24% of the entire population, so there may be a benefit for some models to apply SMOTE and under-sampling. This is implemented in our preprocessing using the imblearn package, of which we are using the SMOTENC (for datasets including categorical features) and RandomUnderSampler methods. For the rest of the report, SMOTE will refer to the combination of synthetic over-sampling and random under-sampling the training set.

The second optional step is the one-hot encoding step. Given our large number of categorical features, many of which are non-binary and not ordinal, some(?) classifiers require this step in order to interpret the data correctly. Each label in a categorical variate (except for the baseline) is turned into a new binary variate representing the existence of that label. This increases the number of features from 19 to 30.

The third optional step is feature scaling. For our three numeric features, all three are non-negative and unbounded, and some models perform better(?) with bounded numerical values. Hence, the numeric features are rescaled to be between 0 and 1 inclusive.

All three optional preprocessing steps are applied independently to the training and testing sets to ensure no leakage occurs. Additionally, all preprocessing steps and models are seeded whenever possible to ensure reproducibility and deterministic behavior.

Decision Tree

The Decision Tree model is implemented with sklearn’s DecisionTreeClassifier, and using one-hot encoded data. First, we tune the hyperparameter α , the tuning parameter used in cost-complexity pruning.

Using k-fold cross-validation with $k=5$, the trees were fully grown using recursive binary splitting, and then repeatedly pruned back to obtain a sequence of subtrees. Then, the average test error (on the unused fold) is computed as a function of α , and the α which maximizes the average test error is chosen.

Then, a Decision Tree is fit on the entire training set, then pruned back using α to return the final decision tree model.

Two versions of this model are fitted; one using the training set, and another using the training set returned from SMOTE.

Random Forest

The Random Forest model is implemented with sklearn’s RandomForestClassifier, and using one-hot encoded data. First, we tune the hyperparameter d , the maximum depth of each tree in the forest.

Using k-fold cross-validation with $k=5$, we used sklearn’s GridSearchCV to find the best value of d which maximized expected test accuracy. Each forest consists of 100 trees, and we elected not to tune the number of random features at each split, selecting the square root of the total number of features.

Then, a Random Forest is fit on the entire training set, again using 100 trees, the square root of the number of features, and using the best d chosen in the previous step. This is then returned as the final random forest model.

Two versions of this model are fitted; one using the training set, and another using the training set returned from SMOTE.

Neural Network (Multi-Layer Perceptron)

The Neural Network model is implemented with sklearn’s MLPClassifier (Multi-Layer Perceptron Classifier), and using one-hot encoded and scaled data.

First, using k-fold cross-validation with $k=5$, we used sklearn’s GridSearchCV to tune the hyperparameters for activation function, solver (optimization algorithm), and hidden layer sizes, evaluated on expected. On the first pass, a wide range of hidden layer sizes is provided to GridSearchCV.

GridSearchCV returns the optimal parameters from the parameters provided to maximize the expected accuracy. Since there are a huge number of possible hidden layer sizes (widths and depths), the first pass only selects a few which are far apart. Then, the activation function and solver method are fixed, and we repeat tuning the hidden layer sizes hyperparameter multiple times, each time narrowing the search based on the optimal parameters returned from the last iteration. A detailed breakdown of one such repeated hyperparameter tuning step is included as a multi-line comment in the code, which can be found in the appendix.

Finally, a Multi-Layer Perceptron is fit on the entire training set with the chosen hyperparameters to be the final model. Two versions of this model are fitted; one with the number of hidden layers fixed to 1 (i.e. a single layer perceptron), and another without this restriction.

Results

- Compare performance
- Key graphs go here
- Discuss performance, significant predictors

Discussion

- Discuss existing Kaggle models on this dataset
- Strengths and limitations of models
- Future research

Appendix

References

Chawla, N. V., K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer. 2002. "SMOTE: Synthetic Minority over-Sampling Technique." *Journal of Artificial Intelligence Research* 16 (June): 321–57. <https://doi.org/10.1613/jair.953>.

Code

Dataset

General purpose Dataset class to unify the loading, preprocessing, and evaluations of the different models.

```
from imblearn.over_sampling import SMOTENC
from imblearn.under_sampling import RandomUnderSampler
from imblearn.pipeline import make_pipeline
from sklearn.compose import make_column_transformer
from sklearn.metrics import accuracy_score, f1_score
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, OneHotEncoder
import pandas as pd

class Dataset:
    """
    Creates a unified way to pre-process, split, and evaluate models on the IBM
    Telco dataset.
    """
    def __init__(self, excel="Telco_customer_churn.xlsx", onehot=False,
                 scale=False, smote=False):
        # Load data
        df = pd.read_excel(excel)
        self.scaler = MinMaxScaler()

        # Remove unrelated columns
        df.drop(columns=[
            "CustomerID", "Count", "Country", "State", "City", "Zip Code",
```

```

        "Lat Long", "Latitude", "Longitude", "Churn Label", "Churn Score",
        "CLTV", "Churn Reason"
    ], inplace=True)

    # Remove rows with empty values
    df = df.loc[df["Total Charges"].str.strip() != ""]

    # Numeric columns
    num_cols = {
        "Tenure Months": int,
        "Monthly Charges": float,
        "Total Charges": float
    }
    df = df.astype(num_cols)

    # Isolate Features and Response
    X = df.drop(columns=["Churn Value"])
    Y = df["Churn Value"]

    # Train/test split
    self.X_train, self.X_test, self.y_train, self.y_test = train_test_split(
        X, Y, test_size=0.1, random_state=441
    )

    # SMOTE (Synthetic Minority Over-sampling TEchnique)
    if smote:
        cat_mask = [not col in num_cols for col in X.columns]
        pipeline = make_pipeline(
            SMOTENC(cat_mask, sampling_strategy=0.5, random_state=441),
            RandomUnderSampler(sampling_strategy=0.5, random_state=441)
        )
        self.X_train, self.y_train = pipeline.fit_resample(
            self.X_train, self.y_train
        )

    # One-hot encoding
    if onehot:
        self.X_train = pd.get_dummies(self.X_train, drop_first=True)
        self.X_test = pd.get_dummies(self.X_test, drop_first=True)
        # cat_cols = [col for col in X.columns if col not in num_cols]
        # transformer = make_column_transformer(
        #     (OneHotEncoder(drop='if_binary'), cat_cols),
        #     remainder='passthrough'
        # )
        # self.X_train = pd.DataFrame(
        #     transformer.fit_transform(self.X_train),
        #     columns=transformer.get_feature_names_out()
        # )
        # self.X_test = pd.DataFrame(
        #     transformer.fit_transform(self.X_test),
        #     columns=transformer.get_feature_names_out()
        # )

    # Scale numeric columns to [0, 1]
    if scale:
        num_col_names = list(num_cols.keys())
        self.X_train[num_col_names] = self.scaler.fit_transform(
            self.X_train[num_col_names]

```

```

    )
    self.X_test[num_col_names] = self.scaler.fit_transform(
        self.X_test[num_col_names]
    )

def get_training_set(self):
    return self.X_train, self.y_train

def get_testing_set(self):
    return self.X_test

def accuracy(self, y_pred, train=False):
    if train:
        return accuracy_score(self.y_train, y_pred)
    else:
        return accuracy_score(self.y_test, y_pred)

def f1(self, y_pred, train=False):
    if train:
        return f1_score(self.y_train, y_pred)
    else:
        return f1_score(self.y_test, y_pred)

def acc_f1(self, y_pred, train=False):
    return (
        round(self.accuracy(y_pred, train=train), 4),
        round(self.f1(y_pred, train=train), 4)
    )

def save_predictions(self, filename, y_pred):
    pd.DataFrame(y_pred, columns=["y_pred"]) \
        .to_csv(f"{filename}.csv", index=False)

```

Decision Tree

```

from collections import namedtuple

from sklearn.model_selection import KFold
from sklearn.tree import DecisionTreeClassifier

from telco import Dataset

def calibrate(ds: Dataset, n_splits=5, silent=True):
    """
    Given a Dataset, returns the value of alpha which maximizes the expected
    testing accuracy for a Decision Tree using k-fold CV.
    """
    # Get data, split into k=nsplits folds
    X, Y = ds.get_training_set()
    kf = KFold(n_splits=n_splits, shuffle=True, random_state=441)

    # Store fold results
    FoldResults = namedtuple("FoldResults", [
        "ccp_alphas", "impurities", "node_counts", "depth",
        "train_scores", "test_scores"
    ])

```

```

])
folds = []

# k-fold CV
for train, test in kf.split(X):
    if not silent:
        print("Testing...")
    X_train, X_test = X.iloc[train], X.iloc[test]
    y_train, y_test = Y.iloc[train], Y.iloc[test]

    # Create Decision Tree, use Cost Complexity Pruning
    clf = DecisionTreeClassifier(random_state=441)
    path = clf.cost_complexity_pruning_path(X_train, y_train)
    ccp_alphas, impurities = path.ccp_alphas[:-1], path.impurities[:-1]

    # Train a DTree with each alpha
    clfs = []
    for ccp_alpha in ccp_alphas:
        clf = DecisionTreeClassifier(
            random_state=441, ccp_alpha=ccp_alpha)
        clf.fit(X_train, y_train)
        clfs.append(clf)

    # Compute metrics
    node_counts = [clf.tree_.node_count for clf in clfs]
    depth = [clf.tree_.max_depth for clf in clfs]
    train_scores = [clf.score(X_train, y_train) for clf in clfs]
    test_scores = [clf.score(X_test, y_test) for clf in clfs]

    # Save fold results
    folds.append(FoldResults(
        ccp_alphas, impurities, node_counts, depth, train_scores,
        test_scores
    ))

# Compute average testing accuracy for all alphas
all_alphas = sorted(set(
    sum([list(folds[i].ccp_alphas) for i in range(n_splits)], []))
))

avg_test_scores = []
a_ind = [0 for _ in range(n_splits)]
for alpha in all_alphas:
    a_sum = 0
    for i in range(n_splits):
        while a_ind[i] + 1 < len(folds[i].ccp_alphas):
            if folds[i].ccp_alphas[a_ind[i] + 1] < alpha:
                a_ind[i] += 1
            else:
                break
        a_sum += folds[i].test_scores[a_ind[i]]
    avg_test_scores.append(a_sum/n_splits)

# Compute best alpha
max_test = max(avg_test_scores)
where_max = avg_test_scores.index(max_test)
best_alpha = all_alphas[where_max]
if not silent:

```

```

        print(max_test, where_max, best_alpha)

    return best_alpha

def create_tree(load=False, smote=False, save=False):
    """
    Given a dataset, tunes or loads a single Decision Tree.
    """
    ds = Dataset(onehot=True, smote=smote)

    if not load:
        # Tune hyperparameter alpha using k-fold CV
        alpha = calibrate(ds, silent=False)
    elif smote:
        # Load tuned value for smote=True
        alpha = 0.0008973247358753291
    else:
        # Load tuned value for smote=False
        alpha = 0.0007307378011487363

    clf = DecisionTreeClassifier(random_state=441, ccp_alpha=alpha)
    clf.fit(*ds.get_training_set())
    y_pred = clf.predict(ds.get_testing_set())
    acc = ds.accuracy(y_pred)
    f1 = ds.f1(y_pred)

    print(clf.get_n_leaves())
    print(clf.get_depth())

    if save:
        ds.save_predictions("dtree", y_pred)

    # 0.8097, 0.5786 (smote=False)
    # 0.8239, 0.6457 (smote=True)
    return (round(acc, 4), round(f1, 4))

print(create_tree(load=False, smote=True))

```

Random Forest

```

from sklearn.model_selection import GridSearchCV
from sklearn.ensemble import RandomForestClassifier

from telco import Dataset

def create_random_forest(load=True, smote=False, save=False):
    """
    Given a dataset, tunes or loads a 100-tree random forest
    """
    ds = Dataset(onehot=True, scale=True, smote=smote)

    if not load:
        param_grid = [{
            'max_depth': list(range(1, 16))

```

```

    }]
    clf = GridSearchCV(
        RandomForestClassifier(oob_score=True, n_jobs=-1, random_state=441),
        param_grid, cv=5, scoring=['accuracy', 'f1'], refit='accuracy'
    )
elif smote:
    # Loads the best parameters from the search for smote=True
    clf = RandomForestClassifier(
        max_depth=11, oob_score=True, n_jobs=-1, random_state=441
    )
else:
    # Loads the best parameters from the search for smote=False
    clf = RandomForestClassifier(
        max_depth=8, oob_score=True, n_jobs=-1, random_state=441
    )

clf.fit(*ds.get_training_set())

if not load:
    print(clf.best_params_)

y_pred = clf.predict(ds.get_testing_set())
acc = ds.accuracy(y_pred)
f1 = ds.f1(y_pred)

if save:
    ds.save_predictions("slp", y_pred)

# 0.8338, 0.6214 (d=8, smote=False)
# 0.8253, 0.6496 (d=11, smote=True)
return (round(acc, 4), round(f1, 4))

print(create_random_forest(load=False, smote=True))

```

Neural Network (Multi-layer Perceptron)

```

from sklearn.model_selection import GridSearchCV
from sklearn.neural_network import MLPClassifier

from telco import Dataset

"""
Hyperparameter tuning for single-layer, smote=False
1. param_grid = [{
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['lbfgs', 'sgd', 'adam'],
    'hidden_layer_sizes': [
        (25,), (50,), (75,), (100,), (125,), (150,), (200,), (250,)
    ]
}]
Selected: 'identity', 'adam', (25,)

2. param_grid = [{
    'hidden_layer_sizes': [
        (5,), (15,), (25,), (35,), (45,)
    ]
}]

```



```

    ]
}]
Selected: (5,)

3. param_grid = [{
    'hidden_layer_sizes': [
        (2,), (4,), (6,), (8,), (10,), (12,), (14,)
    ]
}]
Selected: (4,)

4. param_grid = [{
    'hidden_layer_sizes': [
        (3,), (4,), (5,), (6,), (7,)
    ]
}]
- Set to maximize f1 score, since accuracy is the same
Selected: (5,)

Similar methods for other models.
"""

def create_single_layer_nnet(load=True, smote=False, save=False):
    """
    Given a dataset, tunes or loads a single layer perceptron which maximizes
    the expected prediction accuracy. Hyperparameters are tuned using
    k-fold CV.
    """
    ds = Dataset(onehot=True, scale=True, smote=smote)
    if not load:
        # WARNING: TAKES MANY HOURS
        param_grid = [{
            # 'activation': ['identity', 'logistic', 'tanh', 'relu'],
            # 'solver': ['lbfgs', 'sgd', 'adam'],
            'hidden_layer_sizes': [
                # (25,), (50,), (75,), (100,), (125,), (150,), (200,), (250,)
                # (5,), (10,), (15,), (20,), (25,)
                # (2,), (4,), (6,), (8,), (10,), (12,), (14,)
                (3,), (4,), (5,), (6,), (7,)
            ]
        }]
        clf = GridSearchCV(
            MLPClassifier(
                activation='logistic', solver='adam',
                random_state=441, max_iter=1000
            ),
            param_grid, cv=5, scoring=['accuracy', 'f1'], refit='accuracy'
        )
    else:
        # Loads the best parameters from the search
        clf = MLPClassifier(
            hidden_layer_sizes=(5,), activation='logistic', solver='adam',
            random_state=441, verbose=True, max_iter=1000
        )

    clf.fit(*ds.get_training_set())

```

```

if not load:
    print(clf.best_params_)

y_pred = clf.predict(ds.get_testing_set())
acc = ds.accuracy(y_pred)
f1 = ds.f1(y_pred)

if save:
    ds.save_predictions("slp", y_pred)

# 0.8352, 0.6527 (logi-adam, (25,))
# 0.8381, 0.6607 (logi-adam, (5,)) <- selected
# 0.8381, 0.6587 (logi-adam, (4,))
return (round(acc, 4), round(f1, 4))

def create_multi_layer_nnet(load=True, smote=False, save=False):
    """
    Given a dataset, tunes or loads a multi-layer perceptron which maximizes
    the expected prediction accuracy. Hyperparameters are tuned using
    k-fold CV.
    """
    ds = Dataset(onehot=True, scale=True, smote=smote)
    if not load:
        # WARNING: TAKES MANY HOURS
        param_grid = [{
            'hidden_layer_sizes': [
                # (5, 5), (10, 10), (5, 5, 5), (10, 10, 10)
                (5, 5, 5), (5, 5, 5, 5, 5), (5, 10, 10, 10, 5)
            ]
        }]

        clf = GridSearchCV(
            MLPClassifier(
                activation='logistic', solver='adam',
                random_state=441, max_iter=1000
            ),
            param_grid, cv=5, scoring='accuracy'
        )
    else:
        # Loads the best parameters from the search
        clf = MLPClassifier(
            hidden_layer_sizes=(5,), activation='logistic', solver='adam',
            random_state=441, verbose=True, max_iter=1000
        )

    clf.fit(*ds.get_training_set())

    if not load:
        print(clf.best_params_)

    y_pred = clf.predict(ds.get_testing_set())
    acc = ds.accuracy(y_pred)
    f1 = ds.f1(y_pred)

    if save:
        ds.save_predictions("mlp", y_pred)

```

```
# 0.8338, 0.6465 (logi-adam, (5, 5, 5))  
return (round(acc, 4), round(f1, 4))  
  
print(create_single_layer_nnet(load=False))  
print(create_multi_layer_nnet(load=False))
```