2025

Tarea 3: Clasificación

Santiago Martínez Novoa
202112020
Diego Alejandro González Vargas
202110240

ISIS-4221. PROCESAMIENTO DE LENGUAJE NATURAL
UNIVERSIDAD DE LOS ANDES

Contenido

1. Introducción

El presente informe se realiza con el fin de dar a conocer la forma de implementación propuesta por el grupo para un modelo de generación de textos basada en N-gramas. En este sentido, se presentan cada una de las etapas propuestas por el enunciado para construir el modelo. En primer lugar, se presenta la consolidación de los archivos consolidados para cada uno de los datasets entregados. Posteriormente, se enseñan las tareas de tokenización y normalización de cada una de las frases y palabras que componen los archivos consolidados para cada dataset. En tercer lugar, se enseñan las tareas realizadas para crear los nuevos datasets a partir de los tokens extraídos: el de entrenamiento y el de testeo. Luego de esto, se presenta como se construyeron los modelos para unigramas, bigramas, y trigramas y el formato en que se decidió almacenarlos. Finalmente, se expone un segundo cuaderno de Python con los métodos de testeo de estos modelos: Cálculo de perplejidad y resultados de ejercicios de generación de textos basados en las probabilidades almacenadas en cada uno de los modelos.

2. Implementación de NB & LR (20N)

En este punto se describen las implementaciones realizadas para la clasificación de una serie de noticias de acuerdo con la categoría a la que pertenecen. Para esto, se utilizaron las recomendaciones de utilización de métodos y módulos de la librería scikit-learn de Python.

2.1. Construcción del pipeline y justificación

Para el caso de la consolidación de la creación de las implementaciones propuestas de Naive-Bayes y Regresión Logística, se propuso la utilización del siguiente Pipeline, que permitiese generar las divisiones de los datos de acuerdo con los porcentajes propuestos, así como la evaluación como las representaciones vectoriales de tf y tfidf, cada uno con su herramienta correspondiente de scikit learn.

2.1.1. Carga de datos

Para el proceso de carga de datos, se propuso la utilización del método load_files de la librería scikit-learn de Python. Para esto, se estableció la ruta donde se encontraban todas las carpetas con las “noticias” de las categorías establecidas. Además, entre los demás parámetros se encuentran, el encoding, establcido en latín-1 para evitar que el pipeline se reviente por no reconocer algún símbolo, que se encuentra aun mas blidada con el argumento de ignorar símbolos que arrojen error. Finalmente, se establece una mezcla inicial de los datos como primer método para evitar overfitting

```
def cargar_dataset(dataroot: Path):
    """
```

```
    Carga el dataset 20newsgroups desde archivos organizados en subcarpetas.
    """
    dataset = load_files(
        container_path=str(dataroot),
        encoding="latin-1",
        decode_error="ignore",
        shuffle=True,
        random_state=RANDOM_STATE
    )
    return dataset
```

2.1.2. Particiones de Validación, Entrenamiento y Testeo

Para este punto, se utilizó nuevamente un método de la librería scikit-learn de Python, en este caso, train_test_split. En consecuencia, y como su nombre lo indica, se tenía el problema de que este método dividía los datos en solo 2 conjuntos, y no en 3 como era el requerimiento del enunciado. Así las cosas, se decidió utilizar dos veces el método y con unas constantes establecidas al inicio del cuaderno para establecer, en esa segunda iteración, que se respetaran los porcentajes dichos en el enunciado

```
def partir_train_val_test(X, y, test_size: float, val_ratio_within_trainval:
float, random_state: int):
    """
    Crea partición 60/10/30 estratificada.
    """
    # Primero: (train+val)=70% y test=30%
    X_trainval, X_test, y_trainval, y_test = train_test_split(
        X, y, test_size=test_size, random_state=random_state, stratify=y
    )
    # Luego: dentro del 70%, separa val=10% absoluto y train=60% absoluto
    X_train, X_val, y_train, y_val = train_test_split(
        X_trainval, y_trainval,
        test_size=val_ratio_within_trainval,
        random_state=random_state,
        stratify=y_trainval
    )
    return X_train, y_train, X_val, y_val, X_test, y_test
```

2.1.3. Construir Vectorizadores

En el enunciado se establece que, para cada una de las implementaciones de los clasificadores propuestos, se debía utilizar 2 aproximaciones: por TF o por TFIDF. Para poder dar cumplimiento a este requerimiento, se construyó una función de ayuda que recibiese como parámetro el tipo de vectorizador que se quería implementar en un momento dado, de acuerdo con eso, se utilizaban los métodos de scikit-learn CountVectorizer y TfidfVectorizer para poder dar cumplimiento. Adicionalmente, dado que para la evaluación de 10FOLD se presentan escenarios con alto consumo de

recursos computacional, se construyó un parámetro que permitiese advertir de la limitación de recursos y cargue una parcialidad de los datos extraídos.

```
def construir_vectorizador(kind: str, use_english_stopwords: bool,
optimized=True):
    """
    Construye vectorizador con parámetros optimizados o básicos.
    """
    stop_words = "english" if use_english_stopwords else None
    if optimized:
        # Parámetros menos restrictivos para mejor rendimiento
        min_df = 2
        max_df = 0.95
        max_features = 50000
    else:
        # Parámetros más restrictivos para velocidad
        min_df = 5
        max_df = 0.90
        max_features = 20000
    if kind == "tf":
        return CountVectorizer(
            stop_words=stop_words,
            min_df=min_df,
            max_df=max_df,
            max_features=max_features
        )
    elif kind == "tfidf":
        return TfidfVectorizer(
            stop_words=stop_words,
            min_df=min_df,
            max_df=max_df,
            max_features=max_features
        )
    else:
        raise ValueError("kind debe ser 'tf' o 'tfidf'")
```

2.1.4. Entrenamiento de modelos

Una vez más, para esta sección del código se utilizó la recomendación de scikitLearn para implementaciones pre fabricadas de regresión logística y NaiveBayes multinomial, como se ve en el siguiente snippet:

```
pipelines = {
    "NB + TF": Pipeline([
        ("vec", construir_vectorizador("tf", USE_ENGLISH_STOPWORDS,
optimized=True)),
        ("clf", MultinomialNB())
    ]),
```

```
    "NB + TF-IDF": Pipeline([
        ("vec", construir_vectorizador("tfidf", USE_ENGLISH_STOPWORDS,
optimized=True)),
        ("clf", MultinomialNB())
    ]),
    "LR + TF": Pipeline([
        ("vec", construir_vectorizador("tf", USE_ENGLISH_STOPWORDS,
optimized=True)),
        ("clf", LogisticRegression(solver="liblinear", multi_class="ovr",
random_state=RANDOM_STATE, max_iter=2000))
    ]),
    "LR + TF-IDF": Pipeline([
        ("vec", construir_vectorizador("tfidf", USE_ENGLISH_STOPWORDS,
optimized=True)),
        ("clf", LogisticRegression(solver="liblinear", multi_class="ovr",
random_state=RANDOM_STATE, max_iter=2000))
    ])
}
```

2.1.5. Evaluación de modelos

Finalmente, para la evaluación de cada uno de los modelos construidos, se utilizaron las herramientas de scikit learn para sacar estadísticas del estilo de matriz de confusión enseñadas en el curso. Así mismo, y a partir del cambio de los parámetros de estas funciones pre fabricadas, se encontraron dichas estadísticas a nivel micro y macro, como muestra el siguiente snippet:

```
def evaluar_modelo_simple(pipeline, X_train, y_train, X_val, y_val, X_test,
y_test,
                          nombre_modelo, target_names, mostrar_detalles=True):
    """
    Entrena y evalúa un modelo en validación y test.
    """
    # Entrenar solo en train
    pipeline.fit(X_train, y_train)
    # Evaluar en validación
    y_pred_val = pipeline.predict(X_val)
    acc_val = accuracy_score(y_val, y_pred_val)
    p_mac_val, r_mac_val, f1_mac_val, _ =
precision_recall_fscore_support(y_val, y_pred_val, average="macro")
    p_mic_val, r_mic_val, f1_mic_val, _ =
precision_recall_fscore_support(y_val, y_pred_val, average="micro")
    # Re-entrenar con train+val para test
    X_train_full = np.concatenate([X_train, X_val])
    y_train_full = np.concatenate([y_train, y_val])
    pipeline.fit(X_train_full, y_train_full)
    # Evaluar en test
    y_pred_test = pipeline.predict(X_test)
```

```
    acc_test = accuracy_score(y_test, y_pred_test)
    p_mac_test, r_mac_test, f1_mac_test, _ =
precision_recall_fscore_support(y_test, y_pred_test, average="macro")
    p_mic_test, r_mic_test, f1_mic_test, _ =
precision_recall_fscore_support(y_test, y_pred_test, average="micro")
```

2.2. Evaluación Cruzada & modelo de 10 cortes

2.2.1. Explicación y Conceptos

La validación cruzada es una estrategia estadística que se utiliza para estimar el desempeño real de un modelo de aprendizaje supervisado sin depender de una única partición de los datos. La idea central consiste en dividir el conjunto de entrenamiento en varias partes, llamadas folds o pliegues, y entrenar el modelo repetidamente usando algunas de esas partes para aprender y las restantes para validar. Al rotar estas particiones se logra que cada observación del conjunto de datos participe tanto en fases de entrenamiento como en fases de validación, lo que reduce la varianza de la estimación y da una medida más estable de la capacidad de generalización del modelo.

En la práctica, la técnica se emplea para comparar algoritmos, ajustar hiperparámetros y evitar que los resultados estén sesgados por una división particular de los datos. Una de sus variantes más usadas es la k-fold cross validation, en la que se divide el conjunto en k partes de tamaño similar, se hacen k entrenamientos independientes y luego se promedian las métricas obtenidas. Esta estrategia es especialmente útil cuando no se dispone de un volumen de datos demasiado grande, ya que permite aprovechar mejor toda la información disponible y obtener una evaluación más confiable del rendimiento esperado en datos no vistos.

2.2.2. Implementación

Para poder hacer la implementación de este tipo de evaluación, se juntaron los datasets de validación y entrenamiento, y se utilizaron las funcionalidades de scikit learning para realizar las 10 iteraciones que se necesitan, en especifico StratifiedKFold, y cross_validate. Finalmente, se utilizan el output del segundo mencionado para sacar las estadísticas de interés para la evaluación y posterior comparación:

```
X_trainval = list(X_train) + list(X_val)
y_trainval = list(y_train) + list(y_val)
print(f"   - Datos para CV: {len(X_trainval):,} documentos")
# Cofigurar CV
cv_strategy = StratifiedKFold(n_splits=10, shuffle=True,
random_state=RANDOM_STATE)
scoring_metrics = ["accuracy", "precision_macro", "recall_macro", "f1_macro",
                   "precision_micro", "recall_micro", "f1_micro"]
# CV para Naive Bayes
print(f"\n{'='*60}")
print(" VALIDACIÓN CRUZADA - NAIVE BAYES")
```

```
print(f"{'='*60}")
nb_results = {}
for vectorizer_name, vectorizer_type in [("TF", "tf"), ("TF-IDF", "tfidf")]:
    print(f"\n  Evaluando NB + {vectorizer_name}...")
    pipeline_nb = Pipeline([
        ("vec", construir_vectorizador(vectorizer_type, USE_ENGLISH_STOPWORDS,
optimized=False)),
        ("clf", MultinomialNB())
    ])
    cv_results = cross_validate(
        pipeline_nb, X_trainval, y_trainval,
        cv=cv_strategy, scoring=scoring_metrics, n_jobs=-1
    )
    nb_results[f"NB + {vectorizer_name}"] = cv_results
    print(f" Resultados NB + {vectorizer_name} (10-fold CV):")
    for metric in scoring_metrics:
        scores = cv_results[f"test_{metric}"]
        print(f"   {metric:15s}: {np.mean(scores):.4f} ±
{np.std(scores):.4f}")
```

2.2.3. Reporte de resultados:

VALIDACIÓN CRUZADA - NAIVE BAYES

Resultados NB + TF (10-fold CV):

accuracy : 0.8582 ± 0.0095

precision_macro: 0.8738 ± 0.0090

recall_macro : 0.8554 ± 0.0091

f1_macro : 0.8434 ± 0.0098

precision_micro: 0.8582 ± 0.0095

recall_micro : 0.8582 ± 0.0095

f1_micro : 0.8582 ± 0.0095

Resultados NB + TF-IDF (10-fold CV):

accuracy : 0.8799 ± 0.0066

precision_macro: 0.8898 ± 0.0071

recall_macro : 0.8684 ± 0.0068

f1_macro : 0.8680 ± 0.0074

precision_micro: 0.8799 ± 0.0066

recall_micro : 0.8799 ± 0.0066

f1_micro : 0.8799 ± 0.0066

LOGISTIC REGRESSION

Resultados LR + TF (mejor modelo, 10-fold CV):

accuracy : 0.8941 ± 0.0082

precision_macro: 0.8963 ± 0.0083

recall_macro : 0.8915 ± 0.0084

f1_macro : 0.8927 ± 0.0084

precision_micro: 0.8941 ± 0.0082

recall_micro : 0.8941 ± 0.0082

f1_micro : 0.8941 ± 0.0082

Resultados LR + TF-IDF (mejor modelo, 10-fold CV):

accuracy : 0.9105 ± 0.0062

precision_macro: 0.9118 ± 0.0063

recall_macro : 0.9071 ± 0.0065

f1_macro : 0.9084 ± 0.0064

precision_micro: 0.9105 ± 0.0062

recall_micro : 0.9105 ± 0.0062

f1_micro : 0.9105 ± 0.0062

2.2.4. Búsqueda de Hiperparámetros:

Para poder dar cumplimiento a este punto, se realizó un recorrido por las distintas posibilidades de algunos de hiper parámetros configurables en la implementación de regresión logística, específicamente, regularización y el tipo de la misma, como se muestre en el siguiente snippet:

```
param_grid_lr = {
    'clf__C': [0.1, 0.5, 1.0, 2.0, 5.0],  # Regularización
    'clf__penalty': ['l1', 'l2'],          # Tipo de regularización
    'clf__solver': ['liblinear']           # Solver compatible con l1 y l2
}
```

Y los resultados de estas iteraciones, para cada uno de los escenarios de regresión logística, fueron los siguientes:

Mejores parámetros LR + TF:

clf__C: 0.5

clf__penalty: l2

clf__solver: liblinear

Mejores parámetros LR + TF-IDF:

clf__C: 5.0

clf__penalty: l2

clf__solver: liblinear

2.3. Evaluación de modelos con datos de testeo

2.3.1. Reporte de Métricas:

Evaluación NB + TF en test:

Accuracy: 0.8727 | F1-Macro: 0.8610 | F1-Micro: 0.8727

Evaluación NB + TF-IDF en test:

Accuracy: 0.8911 | F1-Macro: 0.8801 | F1-Micro: 0.8911

Evaluación LR + TF en test:

Accuracy: 0.8915 | F1-Macro: 0.8898 | F1-Micro: 0.8915

Mejores parámetros: {'clf__C': 0.5, 'clf__penalty': 'l2', 'clf__solver': 'liblinear'}

Evaluación LR + TFIDF en test:

Accuracy: 0.9106 | F1-Macro: 0.9087 | F1-Micro: 0.9106

Mejores parámetros: {'clf__C': 5.0, 'clf__penalty': 'l2', 'clf__solver': 'liblinear'}

2.3.2. ¿Cuál es el mejor modelo?

LR + TF-IDF

Representación: TF-IDF

Hiperparámetros optimizados:

clf__C: 5.0

clf__penalty: l2

clf__solver: liblinear

Reporte detallado del mejor modelo:

Clase	Precision	Recall	F1-score	Support
alt.atheism	0.893	0.9	0.896	240
comp.graphics	0.816	0.863	0.839	292

comp.os.ms-windows.misc	0.828	0.82	0.824	293
comp.sys.ibm.pc.hardware	0.83	0.797	0.813	301
comp.sys.mac.hardware	0.891	0.793	0.839	293
comp.windows.x	0.891	0.88	0.885	298
misc.forsale	0.845	0.844	0.845	292
rec.autos	0.923	0.895	0.909	293
rec.motorcycles	0.973	0.956	0.964	298
rec.sport.baseball	0.978	0.964	0.971	293
rec.sport.hockey	0.973	0.978	0.976	300
sci.crypt	0.983	0.93	0.956	297
sci.electronics	0.878	0.883	0.881	294
sci.med	0.931	0.953	0.942	297
sci.space	0.955	0.938	0.946	296
soc.religion.christian	0.891	0.911	0.901	298
talk.politics.guns	0.972	0.963	0.968	292
talk.politics.mideast	0.972	0.978	0.975	296
talk.politics.misc	0.931	0.927	0.929	283
talk.religion.misc	0.901	0.729	0.806	188
accuracy			0.911	5649
macro avg	0.911	0.907	0.909	5649
weighted avg	0.911	0.911	0.911	5649

3. Análisis de Sentimientos

3.1. Implementación por categoría

3.1.1. Pipeline

Para este punto, se siguió un pipeline muy similar al establecido para el primer punto. Así las cosas, en primer lugar, se realizó una carga de los reviews de los diferentes archivos etiquetados, leyendo los documentos línea por línea como sigue, y creando una estructura de diccionarios para guardar la información de cada categoría, como se muestra en las siguientes 2 funciones:

```
def load_all_domains_by_category(root: Path, categories: List[str]) ->
Dict[str, Dict[str, List]]:
    """
    Carga datos organizados por categoría para análisis individual.
    Retorna: {categoria: {X_pos, y_pos, X_neg, y_neg, X_unl}}
    """
    data = {}
    for cat in categories:
        d = root / cat
        Xp, yp = load_review_file(d / "positive.review", default_label=1)
```

```
        Xn, yn = load_review_file(d / "negative.review", default_label=0)
        Xu, yu = load_review_file(d / "unlabeled.review", default_label=None)
        data[cat] = {
            "X_pos": Xp, "y_pos": yp,
            "X_neg": Xn, "y_neg": yn,
            "X_unl": Xu
        }
        print(f"[{cat}] pos={len(Xp)} neg={len(Xn)} unl={len(Xu)}")
    return data
def load_review_file(path: Path, default_label: Optional[int] = None):
    """
    Lee un .review línea por línea y devuelve (X_dicts, y_labels).
    - Si la línea trae #label#, se usa.
    - Si no trae y default_label no es None, se asigna ese.
    """
    X, y = [], []
    with path.open("r", encoding="utf-8", errors="ignore") as f:
        for line in f:
            line = line.strip()
            if line == "":
                continue
            feats, lbl = parse_review_counts_line(line)
            if lbl is None:
                lbl = default_label
            if feats:
                X.append(feats)
                y.append(lbl)
    return X, y
```

Para definir la etiqueta, se aprovecharon las etiquetas descritas al final de cada línea del documento, como se muestra en la siguiente función de ayuda:

```
def parse_review_counts_line(line: str) -> Tuple[Dict[str, float],
Optional[int]]:
    """
    Devuelve (features_dict, label) donde label es 1=positive, 0=negative o
None si no viene.
    Ignora pares malformados.
    """
    feats = {}
    y = None
    parts = line.strip().split()
    for item in parts:
        if item.startswith("#label#:"):
            raw = item.split(":", 1)[1].strip().lower()
            if raw in ("positive", "pos", "1", "p"):
                y = 1
            elif raw in ("negative", "neg", "0", "n"):
                y = 0
            continue
```

```
        if ":" not in item:
            continue
        tok, cnt = item.split(":", 1)
        try:
            val = int(cnt)
        except ValueError:
            try:
                val = float(cnt)
            except ValueError:
                continue
        if tok in feats:
            feats[tok] = feats[tok] + val
        else:
            feats[tok] = val
    return feats, y
```

Finalmente, para el ejercicio del modelo con lexicones se utilizó el dataset de SentiWordNet 3.0.0 que estaba en el Dropbox, y se cargaron sus datos con la siguiente función de ayuda:

```
def load_sentiwordnet(path: Path) -> Dict[str, Tuple[float, float]]:
    """
    Retorna dict: palabra -> (pos_score_promedio, neg_score_promedio)
    Ignora líneas de comentario que comienzan con '#'.
    """
    pos_acc = defaultdict(float)
    neg_acc = defaultdict(float)
    cnt_acc = defaultdict(int)
    with path.open("r", encoding="utf-8", errors="ignore") as f:
        for line in f:
            if not line or line.startswith("#"):
                continue
            cols = line.strip().split("\t")
            if len(cols) < 5:
                continue
            try:
                pos_score = float(cols[2])
                neg_score = float(cols[3])
            except ValueError:
                continue
            terms = cols[4].split()
            for t in terms:
                lemma = t.split("#", 1)[0].lower()
                pos_acc[lemma] += pos_score
                neg_acc[lemma] += neg_score
                cnt_acc[lemma] += 1
    lex = {}
    for w in cnt_acc:
        lex[w] = (pos_acc[w] / cnt_acc[w], neg_acc[w] / cnt_acc[w])
```

```
    return lex
```

Finalmente, se construyeron los vectorizadores y los pipelines siguiendo los mismos lineamientos del punto anterior. La única diferencia fue que, dado que se trataban de datos en formato de diccionario, para el TF se utilizó DictVectorizer(sparse=True) y para el TFIDF: TfidfTransformer(); como se ejemplifica en el siguiente snnipet:

```
def make_nb_tfidf():
    return Pipeline([
        ("dict", DictVectorizer(sparse=True)),
        ("tfidf", TfidfTransformer()),
        ("nb", MultinomialNB())
    ])
```

3.1.2. Implementación de Lexicones & Features

Respecto de la implementación de lexicones, se utilizó una aproximación de 10 características que parten de los puntajes de positivo y negativo dados por SentiWord 3.0.0, y que se describen a continuación:

I. sum_pos: Suma ponderada de puntuaciones positivas (score_positivo × conteo_token)
II. sum_neg: Suma ponderada de puntuaciones negativas
III. hits_pos: Número de tokens únicos con score positivo > 0
IV. hits_neg: Número de tokens únicos con score negativo > 0
V. ratio_pos: Proporción de tokens con sentiment positivo (hits_pos / total_tokens)
VI. ratio_neg: Proporción de tokens con sentiment negativo
VII. max_pos: Máximo score positivo encontrado en el review
VIII. max_neg: Máximo score negativo encontrado
IX. avg_pos: Promedio de scores positivos de tokens que aparecen en el léxico
X. avg_neg: Promedio de scores negativos de tokens que aparecen en el léxico

3.1.3. Reporte de Resultados

Categoría Books:

Modelo	Accuracy	Precision Macro	Recall Macro	F1 Macro	F1 Micro
NB + TF	0.7650	0.7837	0.7650	0.7611	0.7650
NB + TF-IDF	0.7883	0.8162	0.7883	0.7836	0.7883
LR + TF	**0.8067**	**0.8078**	**0.8067**	**0.8065**	**0.8067**
LR + TF-IDF	0.8033	0.8106	0.8033	0.8022	0.8033
LR + Lexicon(SWN)	0.7100	0.7105	0.7100	0.7098	0.7100
NB + Lexicon(SWN)	0.6467	0.6478	0.6467	0.6460	0.6467

Categoría DVD:

Modelo	Accuracy	Precision Macro	Recall Macro	F1 Macro	F1 Micro
NB + TF	**0.8517**	**0.8550**	**0.8517**	**0.8513**	**0.8517**
NB + TF-IDF	0.8433	0.8489	0.8433	0.8427	0.8433
LR + TF	0.8167	0.8167	0.8167	0.8167	0.8167
LR + TF-IDF	0.8500	0.8506	0.8500	0.8499	0.8500
LR + Lexicon(SWN)	0.6950	0.6950	0.6950	0.6950	0.6950
NB + Lexicon(SWN)	0.6633	0.6722	0.6633	0.6590	0.6633

Categoría Electronics:

Modelo	Accuracy	Precision Macro	Recall Macro	F1 Macro	F1 Micro
NB + TF	0.8283	0.8285	0.8283	0.8283	0.8283
NB + TF-IDF	**0.8450**	**0.8452**	**0.8450**	**0.8450**	**0.8450**
LR + TF	0.8350	0.8350	0.8350	0.8350	0.8350
LR + TF-IDF	0.8300	0.8302	0.8300	0.8300	0.8300
LR + Lexicon(SWN)	0.7450	0.7470	0.7450	0.7445	0.7450
NB + Lexicon(SWN)	0.7067	0.7208	0.7067	0.7019	0.7067

Categoría Kitchen:

Modelo	Accuracy	Precision Macro	Recall Macro	F1 Macro	F1 Micro
NB + TF	0.8700	0.8703	0.8700	0.8700	0.8700
NB + TF-IDF	0.8800	0.8806	0.8800	0.8800	0.8800
LR + TF	**0.8850**	**0.8850**	**0.8850**	**0.8850**	**0.8850**
LR + TF-IDF	0.8833	0.8834	0.8833	0.8833	0.8833
LR + Lexicon(SWN)	0.7417	0.7417	0.7417	0.7417	0.7417
NB + Lexicon(SWN)	0.7367	0.7414	0.7367	0.7354	0.7367

3.1.4. Comparaciones

En primer lugar, se compararon NB vs LR, en donde se identificó que funciona un poco mejor regresión logística, como lo muestra las siguientes estadísticas resumidas:

```
                     F1 Macro Promedio  F1 Macro Std  Accuracy Promedio  \
Logistic Regression             0.8000        0.0611             0.8001
Naive Bayes                     0.7837        0.0788             0.7854
```

Posteriormente, se compara TF vs TFIDF vs LEXICON, identificando que es pronunciadamente peor el lexicón, y que el mejor es TFIDF, como sigue:

```
                          F1 Macro Promedio  F1 Macro Std  Accuracy Promedio  \

TF-IDF                               0.8396        0.0322             0.8404
TF (Term Frequency)                  0.8317        0.0363             0.8323
Lexicon Features                     0.7041        0.0346             0.7056
```

3.1.5. Categoría más difícil

La categoría más difícil de clasificar los reviews es Book, porque es el que tiene el score F1 mas bajo de todas las categorías. Esta comparación tomando el mejor modelo de cada una de las categorías. Como lo muestra las siguientes estadísticas:

```
RANKING DE DIFICULTAD (de más difícil a más fácil):
            Best F1 Macro Best F1 Micro Best Accuracy   Best Model
Books           0.806493      0.806667      0.806667      LR + TF
Electronics     0.844979         0.845         0.845  NB + TF-IDF
DVD             0.851319      0.851667      0.851667      NB + TF
Kitchen            0.885         0.885         0.885      LR + TF

CATEGORÍA MÁS DIFÍCIL: Books
   F1-Macro: 0.8065
   Mejor modelo: LR + TF
```

3.1.6. Características más relevantes por categoría

De acuerdo con los pesos sacados del modelo de regresión logística, se sacaron los siguientes TOP de palabras mas positivas y negativas de las reviews de cada categoría:

```
Books:
  Top Positivas: ['great', 'excellent', 'recommend', 'love', 'wonderful']
  Top Negativas: ['not', 'i', 'no', 'bad', 'was']

DVD:
  Top Positivas: ['great', 'his', 'best', 'excellent', 'love']
  Top Negativas: ['not', 'bad', 'worst', 'no', 'was']

Electronics:
  Top Positivas: ['great', 'price', 'good', 'excellent', 'works']
  Top Negativas: ['not', 'work', 'after', 'poor', 'not_work']

Kitchen:
  Top Positivas: ['great', 'easy', 'love', 'easy_to', 'best']
  Top Negativas: ['not', 'was', 'after', 'disappointed', '<num>']
```

3.2. Implementación Global

3.2.1. Reporte de resultados:

A continuación se presentan los resultados de cada uno de los 6 modelos solicitados:

Modelo	Accuracy	Precision (Macro)	Recall (Macro)	F1-Score (Macro)
LR + TF (Global)	85.29%	85.30%	85.29%	85.29%
LR + TF-IDF (Global)	84.67%	84.76%	84.67%	84.66%
NB + TF-IDF (Global)	84.33%	85.09%	84.33%	84.25%
NB + TF (Global)	83.13%	83.42%	83.13%	83.09%
LR + Lexicon(SWN) (Global)	71.21%	71.27%	71.21%	71.19%
NB + Lexicon(SWN) (Global)	68.75%	69.62%	68.75%	68.40%

3.2.2. Comparaciones

TF VS TFIDF

```
F1 Macro Promedio F1 Macro Std Accuracy Promedio  \
TF-IDF                      0.844525     0.002039              0.845
TF (Term Frequency)          0.84189     0.011015           0.842083
Lexicon Features            0.697944     0.013928           0.699792

                    Precision Promedio Recall Promedio
TF-IDF                        0.849238           0.845
TF (Term Frequency)           0.843619        0.842083
   Lexicon Features              0.704429        0.699792
```

NB VS LR:

	F1 Macro Promedio	F1 Macro Std	Accuracy Promedio
Logistic Regression	0.803781	0.065041	0.803889
Naive Bayes	0.785793	0.072123	0.787361

	Precision Promedio	Recall Promedio	Mejor F1
Logistic Regression	0.804442	0.803889	0.852905
Naive Bayes	0.793748	0.787361	0.842486

¿Vale la pena tener un clasificador por clase?:

SI, se puede ganar casi un 10% de mejora de rendimiento si se utiliza un clasificador por clase

3.2.3. Características mas relevantes:

Para el caso de este modelo global, las características más importantes son las siguientes:

Positivas: ['great', 'excellent', 'love', 'easy', 'best', 'easy_to', 'the_best', 'perfect']

Negativas: ['not', 'bad', 'waste', 'was', 'disappointed', 'poor', 'worst', 'after']

4. Conclusiones

Los experimentos demuestran consistentemente que Logistic Regression supera a Naive Bayes en la mayoría de escenarios evaluados, tanto en la clasificación de newsgroups como en el análisis de sentimientos multi-dominio. Esta ventaja se debe a que LR no asume independencia condicional entre características como NB, permitiéndole capturar mejor las correlaciones existentes en datos textuales de alta dimensionalidad. En el análisis de sentimientos, LR mostró mejores resultados en 3 de 4 categorías, con diferencias promedio de +0.02 a +0.08 en F1-score. La única excepción notable fue con características puramente lexicográficas (SentiWordNet), donde la diferencia se redujo significativamente debido a la menor dimensionalidad y mayor densidad de las características.

TF-IDF demostró ser consistentemente superior a TF simple en todos los experimentos, con mejoras promedio de 0.015-0.025 en F1-score. Esta superioridad se debe a la normalización que aplica TF-IDF, reduciendo el peso de términos muy frecuentes y enfatizando palabras discriminativas para cada clase. Las características basadas en lexicon (SentiWordNet) mostraron rendimientos inferiores a las representaciones estadísticas, obteniendo F1-scores 0.10-0.15 puntos menores que TF-IDF. Esto sugiere que, aunque las características semánticas aportan interpretabilidad, las representaciones estadísticas de bolsa de palabras capturan mejor los patrones discriminativos en tareas de clasificación textual, especialmente cuando se cuenta con suficientes datos de entrenamiento.

El análisis multi-dominio reveló diferencias significativas en la dificultad de clasificación entre categorías. Books resultó ser la categoría más difícil (F1-score máximo: 0.806), mientras que Kitchen fue la más fácil (F1-score máximo: 0.885), con una diferencia de 0.079 puntos. Esta variación se atribuye a diferencias en la complejidad del lenguaje, consistencia del vocabulario y especificidad de los términos utilizados en cada dominio. Electronics y DVD mostraron dificultad intermedia, sugiriendo que algunos dominios tienen patrones lingüísticos más regulares y predecibles que otros, lo cual tiene implicaciones importantes para la selección de estrategias de modelado específicas por dominio.

La comparación entre modelos individuales por categoría versus un modelo global único reveló un trade-off clásico entre especialización y generalización. Los modelos globales, entrenados con datos combinados de todas las categorías, mostraron un rendimiento promedio ligeramente inferior a los modelos especializados (-0.02 a -0.04 en F1-score), pero ofrecen ventajas prácticas significativas: mayor simplicidad de implementación, menor costo computacional de mantenimiento, y mejor capacidad de generalización a nuevos dominios. El modelo global LR + TF alcanzó F1-score de 0.853, superando el rendimiento promedio individual, lo que sugiere que la combinación de datos puede beneficiar la generalización, especialmente cuando se usan representaciones robustas como TF-IDF.

El análisis de los parámetros de Logistic Regression reveló patrones interesantes en las características más discriminativas por categoría. Palabras como "great", "excellent", "love" emergieron consistentemente como indicadores positivos fuertes, mientras que "not", "bad", "waste" fueron los indicadores negativos más potentes. Sin embargo, se observaron especialidades por dominio: Electronics enfatizó términos técnicos como "price", "works", "speakers", mientras que Kitchen privilegió aspectos funcionales como "easy", "clean", "little". Esta especialización léxica justifica parcialmente el uso de modelos por categoría, aunque las características globales mantienen suficiente poder discriminativo para un modelo unificado, especialmente considerando la eficiencia operacional y la escalabilidad que esto representa.